UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Spiller-Muys

Vpeljava arhitekture na podlagi modelov z uporabo odprtokodnih orodij

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE
STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Damjan Vavpotič

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika dela:

V okviru diplomske naloge najprej predstavite standard arhitekture na podlagi modelov (MDA) in preučite prednosti in slabosti omenjene arhitekture. V nadaljevanju predstavite možnosti za vpeljavo razvoja, ki temelji na arhitekturi na podlagi modelov, pa tudi celoten potek takšnega razvoja. Uporabo pristopa in ustreznih odprtokodnih orodij demonstrirajte na dovolj kompleksnem primeru. V zaključku kritično ocenite prednosti in slabosti takšnega pristopa k razvoju.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Matej Spiller-Muys, z vpisno številko **63000274**, sem avtor diplomskega dela z naslovom:

Vpeljava arhitekture na podlagi modelov z uporabo odprtokodnih orodij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Damjana Vavpotiča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 5. julij 2014 Podpis avtorja:

Kazalo

Kazalo slik

Seznam uporabljenih kratic

kratica	angleško	slovensko
MDA	Model drivel arhitecture	arhitektura na podlagi modela
DBMS	database management system	sistem za upravljanje podatkovnih baz
NOSQL	Not only SQL	Ne samo SQL (Ne relacijska podatkovna baza)
UML	unified modeling language	Poenoten jezik modelov
SCXML	W3C State Chart XML	W3C XML diagram stanj
CIM	computation indepedent model	računsko neodvisen model
PIM	platform indepedent model	model neodvisen od okolja
PSM	plaftorm specific model	model prilagojen okolju
xUML	executable UML	izvršjivi UML
POJO	plain java object	navadni java objekti
XML	Extensible Markup Language	razširljiv označevalni jezik
JSTL	Java Standard Tag Library	Javanska standardna knjižnica značk

Povzetek

V diplomski nalogi je predstavljena implementacija arhitekture na podlagi modelov z uporabo odprtokodnih orodij. V prvem delu je predstavljen standard, ki je osnova za postavitev takšne arhitekture. V nadaljevanju so analizirane prednosti in slabosti takšnega pristopa, ter kako jih vpeljati v tradicionalni razvoj programske opreme. V nadaljevanju je apliciran takšen način razvoja na vnaprej določenem problemu. Problem je sestavljen iz poenostavljenega problema iz prakse, in sicer beleženja in reševanja napak pri projektu. Do problema se pristopa z definiranjem modela, ki je razumljiv analitikom. Model je uporabljen kot osnova za izdelavo transformacij, s katerimi se pretvori izvorni model v programsko rešitev. Za realizacijo rešitve so uporabljena odprtokodna orodja, s katerimi je preizkušen opisani pristop na realnem problemu. V zaključku so analizirane dobre prakse in izpostavljene težave, do katerih je prišlo med razvojem rešitve.

Ključne besede: arhitektura, model, programiranje, UML, OMG, MDA.

Abstract

Using open source tools to implement model driven architecture will be presented in this thesis. In first chapter a standard, which is a basis for building of such architecture will be introduced. Advantages and disadvantages of such approach will be analyzed, as well as possibilities of its implementation into traditional software development. In practical part, this development approach will be applied on before defined problem. The problem consists of a simplified case from practice, namely tracking and solving of tasks within a project. An approach chosen for this problem is to define a model, which is understandable to analytics. The model is used as a basis for building transformations, wherewith original model is transformed into a software solution. Open source tools are used for resolving the real problem based on such approach. In conclusion good practices will be analyzed and difficulties that came forth in the process of solution development will be pointed out.

Keywords: architecture, model, programming, design, UML, OMG MDA.

1. Uvod

Že od nekdaj se sprašujemo, kako pohitriti in izboljšati kakovost in hitrost razvoja programske opreme, kako zmanjšati ceno ter povečati konkurenčnost. Zato smo se odločili, da raziščemo in uporabimo v praksi, glede na tradicionalnen način, drugačen način razvoja.

V diplomski nalogi bomo predstavili praktično uporabo MDA načina razvoja s pomočjo odprto kodnih sodobnih orodij.

V prvem poglavju bomo predstavili in razdelali standard MDA. Analizirali bomo prednosti in slabosti MDA standarda. Ukvarjali se bomo z možnostjo vpeljave MDA standarda v različne razvojne procese.

V nadaljevanju bomo analizirali različna odprtokodna orodja, s katerimi se lahko uporablja MDA pristop v celoti ali pa samo kot cel celotnega MDA procesa.

V zadnjem delu bomo izbrali nabor predhodno analiziranih orodij in se ukvarjali z uporabo le teh pri reševanju nekega konkretnega problema. Za konkreten problem smo si izbrali splošen problem beleženja, vodenja in reševanja nalog v podjetjih. Naredili bomo analizo in definirali svoj analitičen model. Ta model bomo nato pretvorili s pomočjo transformacij, ki jih bomo pripravili tekom naloge v zaključeno delujočo rešitev. Rešitev bo sestavljena iz spletne in mobilne aplikacije. Cilj naloge je, da bosta obe aplikaciji zasnovani na podlagi istega modela.

V zaključku bomo predstavili lastne ugotovitve pri uporabi takšnega pristopa. Med drugim bomo analizirali čas, ki je potreben za razvoj, kot tudi razvitost današnjih orodij.

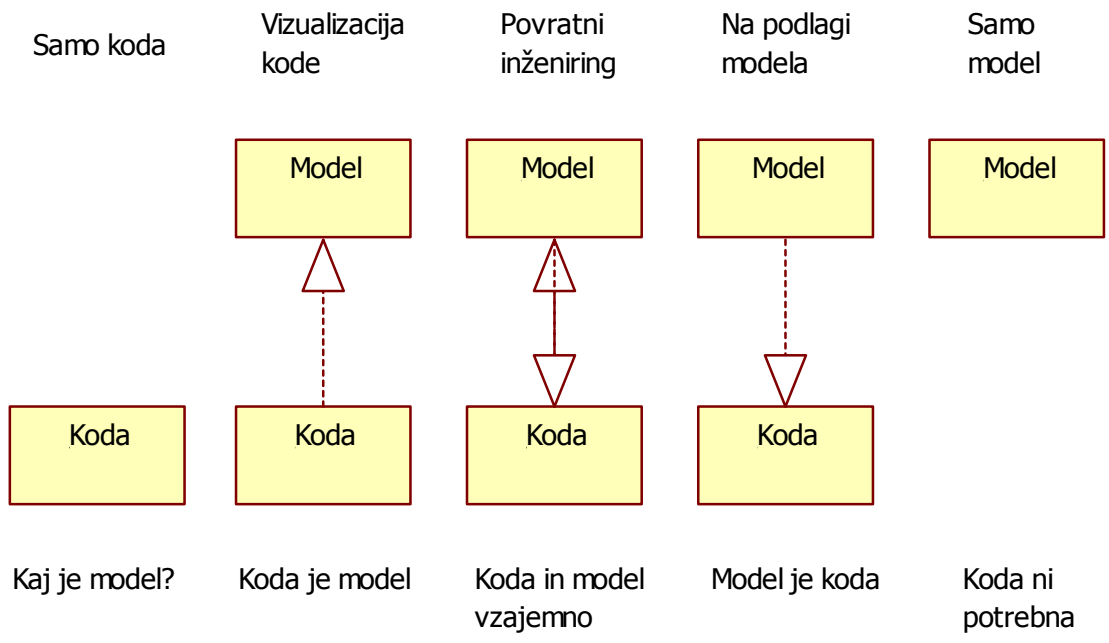
2. Arhitektura na podlagi modela

2.1. Predstavitev

Arhitektura na podlagi modela je način izdelave rešitev na način, kjer se programira na višjem nivoju od današnjih jezikov tretje generacije. Pri klasičnem razvoju se razvojniki sprašujejo, kako nekaj narediti, namesto da bi se spraševali, kaj narediti. Pri izdelavi rešitve na podlagi modela pa dajemo prednost izdelavi različnih modelov, s katerimi opišemo, kaj želimo doseči. V fazi analize se ukvarjamo z izgradnjo modela, ki lahko zelo dobro opisuje rešitev problema. Na podlagi izdelanih modelov nato uporabimo transformacije, s katerimi izdelamo končno rešitev našega problema. Transformacije pretvarjajo model v nov, bolj specifičen model, ali pa nam pretvorijo model v izvorno kodo, ki je že tudi končna rešitev našega problema na izbranem okolju. Pri takšnem pristopu se ukvarjamo s problemom od zgoraj navzdol. To pomeni, da začnemo reševati problem na najvišjem nivoju, nato pa rešujemo problem na nižjih nivojih, in sicer pretvarjamo model v vedno bolj specifično obliko, ki se na vsakem nivoju bolj ukvarja z uporabljenimi tehnologijami in programskimi jeziki. Modeli so sestavljeni iz posameznih elementov. Pri pretvarjanju modela v drugo obliko se s posameznimi deli ukvarjamo kar se da neodvisno. S pomočjo ene transformacije in izvornega modela se osredotočimo na rešitev posameznega dela problema in ga rešujemo samo enkrat, ne glede na kompleksnost in kasnejše vsebinske spremembe v definiranem modelu. S pomočjo transformacij se osredotočimo na rešitev neke osnovne funkcionalnosti modela. Takšen pristop omogoča modularno zasnovo sistema. Prednost modularnega pristopa je reševanje problemov na neodvisen način, ki ga je mogoče neodvisno od celotne rešitve zasnovati, razviti in preverjati. Ker je takšen pristop zelo uporaben, se ga uporablja tudi že v tradicionalnem razvoju. Arhitektura na podlagi modela pa takšen pristop dviga še na višji nivo.

2.2. Tradicionalen razvoj

Tradicionalen razvoj je razvoj programske opreme predvsem s pomočjo jezikov tretje generacije. Tretja generacija jezikov se ne ukvarja več s specifikami strojne opreme, temveč s tem, kako reševati probleme. Izdelava programske opreme je zapletena in draga. Pri izdelavi programske opreme uporabljamo različne tehnologije in komuniciramo z različnimi sistemi. Ker je napredek zelo hiter, je nujno potrebno učenje novih tehnologij. V celotnem življenjskem ciklu programske opreme prihaja do veliko sprememb. Do danes so se razvile različne stopnje razvoja programske opreme.



Slika 1: Oblike klasičnega razvoja

2.2.1. Samo koda

Pri tem načinu smo odvisni samo od kode. Ločeno definiranih modelov ni. Modeli so definirani izključno v kodi v jeziku tretje generacije.

Težave pri takšnem načinu razvoja so v težavnem dojemanju celotne slike ter glavnih karakteristik sistema. Težave se pojavijo pri povečani kompleksnosti predvsem pri daljšem časovnem obdobju, saj prvotni razvijalci niso več na voljo ali pa so že pozabili glavne karakteristike rešitve.

2.2.2. Vizualizacija kode

V tem načinu je glavna informacija še vedno prisotna samo v kodi. S pomočjo orodij želimo predstaviti celotno sliko na višjem nivoju.

Slabost takšnega pristopa je, da je model v celoti odvisen od kode. Predstavitev modela je sestavljena v obliki, kakor je zapisano v kodi, abstrakcija je posledično omejena.

2.2.3. Povratni inženiring

Povratni inženiring se izvaja med kodo in abstraktnim modelom. Abstraktni model predstavlja arhitekturo sistema na višjem nivoju. Slabost takšnega modela je, da orodja večinoma generirajo samo prazne metode, ki jih mora razvijalec sprogramirati ko konca. Ker koda sčasoma postaja vedno bolj zapletena, je potrebno popravljati tudi abstraktni model. Težava pri tem je, da je večina orodij omejena pri pretvarjanju konceptov v kodi v abstraktni model.

2.2.4. Na podlagi modela

S pomočjo tega pristopa imajo modeli zadostno količino informacij, ki so potrebne za celotno implementacijo sistema. Da lahko dosežemo takšen način, mora model vsebovati:

- podatkovni model
- poslovno logiko
- elemente potrebne za predstavitev
- servise in vmesnike.

Slabost tega pristopa je, da so modeli v večini primerov vezani na orodje, ki ga uporabljamo, medtem ko interoperabilnosti med različnimi orodji ni. Z orodji, ki jih uporabljamo, smo prav tako vezani na stil aplikacij, ki jih takšna orodja podpirajo.

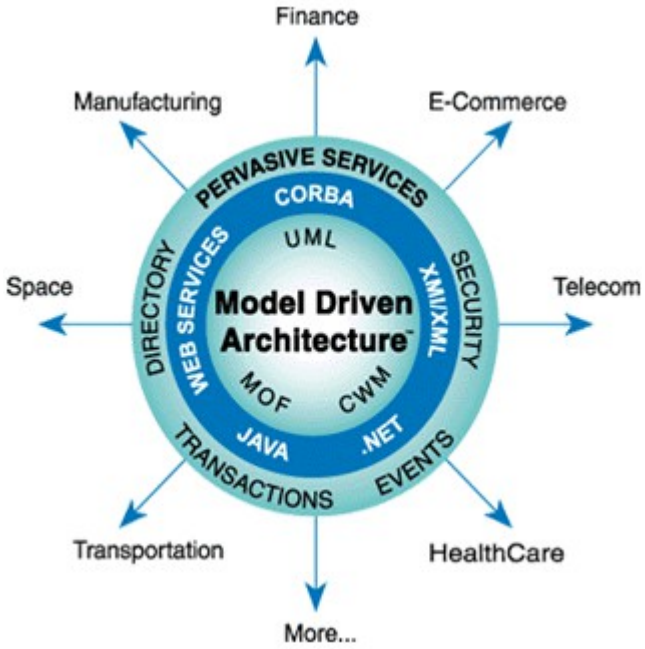
2.2.5. Samo model

Model se uporablja izključno kot pomoč pri razumevanju domene. Pri tej uporabi je model neodvisen od implementacije. Slabost takšnega pristopa je, da je model samo dodatna predstavitev problema, zato je vzdrževanje modela dražje, zaradi česar pogosto pride do situacije, ko se model preneha vzdrževati.

2.3. MDA standard

Arhitekturo na podlagi modela je zasnovala in formalizirala organizacija OMG. OMG je neprofitna organizacija, kateri se lahko priključi vsak in sodeluje pri razvoju standardov prihodnosti. OMG predstavlja način, kako na nevtralen način definirati in realizirati poslovne in tehnološke spremembe.

OMG je do danes razvila že večje število pomembnih specifikacij, kot so CORBA, OMG IDL, IIOP, UML, MOF, XML, CWM OMA in definicije domen za industrije, kot je zdravstvo, proizvodnja, telekomunikacije in podobno. Arhitektura s pomočjo modelov bazira na teh izkušnjah in gradi povezljiva ogrodja za definicijo povezanih sistemov v prihodnosti.



Slika 2: Področja uporabe MDA [8]

2.4. Cilji standarda OMG MDA

OMG MDA formalnen način razvoja je zasnovan na predpostavljenih ciljih, in sicer na takšen način, da zagotavlja določene prednosti glede na klasičen razvoj. Te prednosti morajo biti vidne v celotni verigi razvoja od načrtovanja do vzdrževanja.

Cilji so:

- prenosljivost: funkcionalnost, ki je razvita na MDA način se lahko hitreje in ceneje prilagaja spremembam okoljem in razvojnim platformam.

- produktivnost: zaradi avtomatizacije razvojnih nalog se lahko razvijalci ukvarjajo z reševanjem funkcionalnosti sistema. Zaradi tega se skrajša proces razvoja.

- kvaliteta: zaradi formalne ločitve med problemom in implementacijo problema ter ločevanja med posameznimi nivoji se poveča kvaliteta celotnega sistema.

- integracija: integracija s starimi sistemi se izboljša, saj je integracija definirana s pomočjo abstrakcije v modelu.

- testiranje: modeli se lahko preverjajo že v fazi razvoja samega modela. Modeli se preverjajo na podlagi zahtev naročnika in s pomočjo transformacij na različnih infrastrukturah.

- vzdrževanje: arhitektura je formalno opisana v obliki, ki jo razumejo orodja. Orodja lahko pomagajo pri preverjanju modela. Nadgradnje modela so zaradi višje abstrakcije hitrejše in cenejše.

2.5. Kaj je model?

Model je centralni del MDA načina razvoja. Model je način abstraktne predstavitve nečesa, kar obstaja v realnosti. Pri MDA načinu imamo na voljo več tipov modelov. Ti tipi modelov predstavljajo različne poglede v arhitekturi rešitve.

Pri izdelavi posameznega modela se nam porajajo vprašanja:

Ali je model analitičnen ali načrtovalnen?

Nivo detajlov, ki jih vključuje?

Ali je posloven ali programski model?

Ali model cilja na specifično tehnologijo?

Odgovori na zgornja vprašanja določajo obliko modela. Glede na posamezne potrebe OMG MDA določa 3 tipe modelov, ki so pogledi iz različnih perspektiv. Ti modeli so računsko neodvisen model (CIM model), model neodvisen od okolja (PIM model) in model odvisen od okolja (PSM model).

Posamezen tip modela ni absoluten. V enem kontekstu se ga lahko vidi kot model PIM, v drugem pa kot model PSM. Primer razlikovanja takšnega modela je na primer spletni servis, ki je v problemu tehničen del implementacije in kot tak PSM model, medtem ko je v kontekstu rešitve v neodvisnosti od platforme PIM model.

2.6. Računsko neodvisen model (CIM model)

Računsko neodvisen model je večinoma predstavljen kot poslovnen ali domenski model, ker je narejen v obliki, ki je razumljiva domenskim strokovnjakom. Predstavlja opis problema na način, kjer vprašanje, kako bo rešitev narejena, ter v kateri tehnologiji je narejena rešitev, ni pomembno za razumevanje in definiranje rešitve. Pomembno je samo, kaj je od rešitve pričakovano. Vloga CIM modela je zbližati domenske strokovnjake in tehnične strokovnjake s pomočjo formalne oblike zahtev, katerim je mogoče slediti po celotnem procesu razvoja. Vsaka zahteva v CIM modelu mora biti povezana z realizacijo v PIM ali pa PSM modelu.

2.7. Model neodvisen od okolja (PIM model)

Model je narejen na način, da je neodvisen od okolja, v katerem se bo končna rešitev izvajala. V fazi izdelave modela se ne ukvarjamo z značilnosti okolja, temveč samo z realizacijo poslovnih zahtev, definiranih v CIM modelu.

2.8. Model odvisen od okolja (PSM model)

Model odvisen od okolja določa, kako se bo PIM model apliciral na izbrano okolje, v katerem bo umeščena rešitev. Takšen model odraža vse tehnične podrobnosti izbranih tehnologij. Primer takšnega modela je model, ki vsebuje poleg atributov razredov tudi metode, ki so potrebne za dostop do teh atributov.

2.9. Posamezni nivoji modela

Za razumevanje različnih povezav med različnimi OMG standardi OMG uporablja pri svojih standardih štiri nivojsko arhitekturo. V OMG terminologiji se ti nivoji imenujejo M0, M1, M2, M3.

Prvi nivo je konkreten pogled na nek problem. Gre za izvajajoči sistem, kjer obstajajo entitete modela z realnimi podatki. Na primer uporabnik sistema z imenom Nejc je naredil novo instanco objekta Košarica, ki vsebuje konkretne izdelke, ki jih želi kupiti.

Drugi nivo je model sistema, ki je opisan v prvem nivoju. Ta nivo določa, kako je posamezen objekt prikazan. Opisuje, katere atribute vsebuje, ter kako so različni objekti med seboj povezani.

Tretji nivo določa model od modela drugega nivoja. Modeli na drugem nivoju so realizacije opisa modela na tretjem nivoju. Primer takšnega modela je UML. UML določa, na kakšen način lahko modele na drugem nivoju gradimo. Takšen model imenujemo meta model. Meta model nam olajša pripravo modela, saj določa, katere podatke lahko model drugega nivoja vsebuje.

Četrti nivo določa model od tretjega nivoja. Je osnova za definiranje poljubnih modelov in določa osnovna pravila, kako se model v splošnem definira. V okviru OMG skupine je MOF standardni jezik za definirane kateregakoli modela. MOF definira M3 model. MOF igra podobno vlogo kakor EBNF pri definiciji programskih jezikov. Gre za domensko specifičen jezik, ki se uporablja za definicijo meta modelov.

Fundacija Eclipse ima svojo implementacijo MOF modela, ki je bila razvita neodvisno od MOF, in se imenuje Ecore, ter se uporablja v Eclipse Modeling Framework. Ecore rešitev je podmnožica MOF modela. V verziji MOF 2.0 je vpeljan osnovni MOF, ki je podmnožica popolnega MOF. Osnovni MOF, sicer tako imenovani Essential MOF, je razen v poimenovanju, bolj ali manj enak Ecore.

2.10. Kaj je transformacija

V opisanem MDA procesu so prikazane različne vloge modelov. Vsak izmed modelov PIM, PSM in izvorna koda igrajo eno izmed vlog v MDA procesu. Transformacija med posameznimi stopnjami procesa nam pretvarja en model v drugega. PIM v PSM in PSM v izvorno kodo (slika 3). Ti dve transformaciji sta ključni v MDA razvojnem procesu. Cilj transformacije je pretvorba poljubne realizacije formalne definicije modela.

Primer takšne transformacije je pretvorba iz UML modela v Java izvorno kodo. Cilj te transformacije je, da se jo lahko uporabi na poljubnih UML modelih.

Transformacije je sestavljena iz zbirke transformacijskih pravil, ki določajo, kako se nek del modela pretvori v drug del modela ali izvorne kode.

Slika 3: Transformacije med različnimi tipi modelov

2.11. Vpeljava MDA procesa

Pri MDA procesu razvoja ni vnaprej določeno, na kakšen način bomo razvijali. Pri vseh načinih razvoja je v ospredju model in transformacije modela. Predstavljen bo način vpeljave MDA v dva najbolj razširjena procesa razvoja.

2.11.1. Klasičen model razvoja

Pri klasičnem procesu razvoja (*Waterfall*) je celoten proces razvoja programske opreme razdeljen v več faz, katere si sledijo zaporedno. Po zaključitvi vsake faze se naredi pregled in potrditev zaključene faze. Šele po potrditvi se lahko začne izvajati nova faza. Razvoj je razdeljen na naslednje faze:

- zbiranje zahtev
- analiza
- načrtovanje
- razvoj
- namestitev
- vzdrževanje.

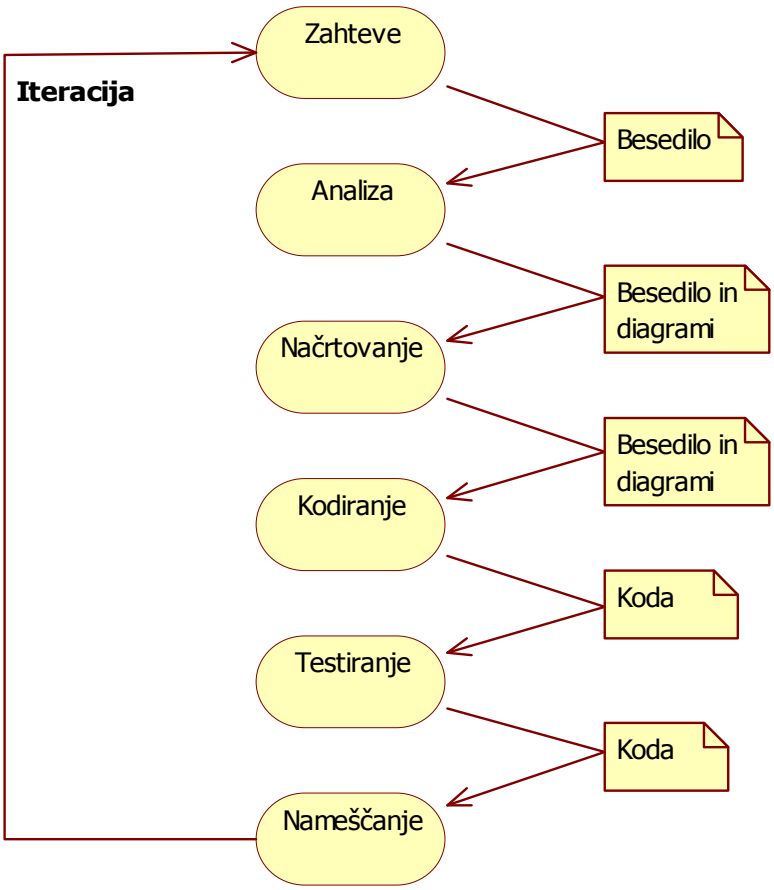
MDA način se lahko vključi že v fazo zbiranja zahtev, kjer se lahko stranki vizualno prikaže realizacijo njihovih zahtev. Analitiki ki zbirajo zahteve, lahko predstavijo zahteve v bolj formalni obliki, ki je lahko zelo dobra osnova za izdelavo modela v kasnejših fazah. V fazi načrtovanja se določijo PSM modeli in transformacije, s pomočjo katerih se transformira model v izvorno kodo.

2.11.2. Agilno programiranje

Agilno programiranje je skupek metod na načelih iterativnega in progresivnega razvoja.

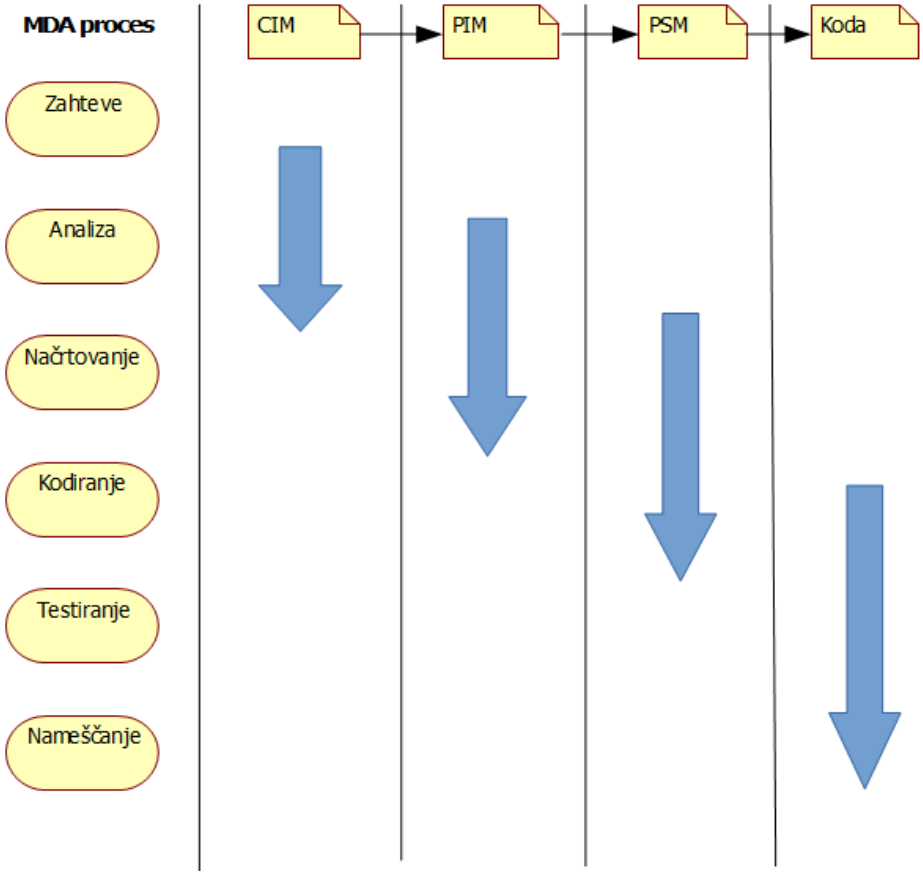
Razvoj se začne kot izdelava prototipa, nato se v vsaki iteraciji razvoja dodaja nove funkcionalnosti in testiranja skozi celoten razvoj. Agilni razvoj promovira prilagodljivo planiranje, kjer se vidi stalen napredek.

Pri spremembi MDA modela se spremeni tudi koda. Ker se pri agilnem pristopu spremembe dogajajo neprestano v vsaki iteraciji, lahko MDA pristop pripomore k hitrejšim implementacijam sprememb. Iterativni razvoj je pred uporabo MDA načina razvoja predstavljen na sliki 4.



Slika 4: Agilni proces

Po vpeljavi MDA načina razvoja je v ospredju definiranje modela in transformacij med prehodi različnih stanj procesa. Glavni namen je avtomatizacija prehodov iz enega v drug proces. MDA način se na prvi pogled od tradicionalnega razvoja bistveno ne razlikuje. Glavna razlika je v rezultatih posameznih korakov, ki so v primeru MDA formalni modeli. CIM, PIM, PSM in koda so rezultati posameznih korakov v razvojnem ciklu. Razvoj se začne na najvišjem abstraktnem nivoju, ki se v vsakem koraku zmanjšuje. To omogoča izdelavo bolj kompleksnih sistemov. Na sliki 5 je razvidno, kako se po vpeljavi MDA načina razvoja razvijajo posamezni tipi modelov. V primeru agilnega razvoja se celoten razvojni proces lahko večkrat ponovi.



Slika 5: Proces po vpeljavi MDA

2.12. Prednosti MDA razvoja

MDA način razvoja je lahko hitrejši zaradi uporabe modela na višjem abstraktnem nivoju, kar pomeni, da en element v modelu proizvede na koncu več vrstic kode. Posledično lahko na višjem nivoju zgradimo več funkcionalnosti v istem času. Zaradi tega se lahko zniža tudi cena razvoja[6].

Kvaliteta se zaradi dodatne abstrakcije povečuje. Vse dobre prakse, ki smo se jih skozi leta naučili pri klasičnem razvoju, lahko prenesemo v transformacije. Tako lahko dobimo boljšo rešitev z manj napakami.

Testiranje rešitve je hitrejše, ni se nam potrebno ukvarjati s posameznimi detajli končne rešitve, temveč lahko preverjamo pravilnost modela že med načrtovanjem.

Model dobro zajame domensko znanje. Ni se potrebno ukvarjati s tehničnimi problemi.

Pri razvoju prihaja do problema neusklajenosti med specifikacijami in rešitvijo. Pri MDA načinu razvoja je dokumentacija vedno usklajena z specifikacijami, saj je model definicija domene. Tudi na tem nivoju sicer obstaja določen del dokumentacije, ki ni vezan na model. Takšen primer dokumentacije so npr. pojasnila, zakaj se je izbrala določena rešitev v PIM modelu.

Pri MDA načinu razvoja imajo večjo vlogo domenski strokovnjaki. Ker je model bližje analitičnemu pristopu, se lahko domenski strokovnjaki ukvarjajo s problemom. S tehnologijo se ni potrebno ukvarjati, zaradi česar se kvaliteta povečuje.

Pri razvoju je prenosljivost pomembna. Glede na to, da je pri MDA razvoj usmerjen na PIM model, imamo lahko prenosljivost doseženo z več kot enim PSM modelom. Nivo doseganja prenosljivosti je odvisen od transformacij, ki pretvarjajo PIM v PSM model.

Pogosto imajo podjetja glede na izkušnje že pripravljeno arhitekturo svojih rešitev. Pri tem je težko preverjati, ali se ta arhitektura tudi uporablja. MDA razvoj jamči, da se takšna arhitektura ohranja skozi vse nivoje programske opreme.

2.13. Slabosti MDA

MDA ima tudi svoje slabosti. MDA način razvoja je danes razširjen v omejeni meri.

Čeprav je tehnično gledano MDA zasnovan kot neodvisen od posamezne implementacije, je dandanes problem pri posameznih implementacijah orodij MDA ta, da med seboj niso združljive. Posledično pride do problema odvisnosti od posameznega ponudnika rešitve.

MDA je zasnovan preveč idealistično zaradi svojega enosmernega načina predstavitve modelov. Modeli, ki vključujejo akcije, se transformirajo delno ali v celoti samo v eno smer. To se sklada z OMG vizijo, kjer se celotna domena preslika v UML model in nato preko transformacije preslika v celotno končno rešitev. Težave, ki pri tem nastopijo, so fine prilagoditve končne rešitve, zaradi česar so pogosto potrebne dodatne naknadne transformacije kode. Primer takšnih prilagoditev so na primer optimizacije podatkovnega modela, kot je definiranje indeksov, ter na primer denormalizacija modela zaradi zmogljivostnih potreb. Zaradi potrebe po generičnosti so takšni modeli pogosto preveč kompleksni in nerazumljivi. Glede na to, da se je MDA način izkazal kot preveč idealističen, so se pojavile bolj pragmatične rešitve, tako imenovane ”*pragmatic MDA*”, ki omogočajo uporabo bolj tradicionalnih mehanizmov, kot je na primer povratni inženiring.

Vizualna podoba rešitve je preveč pusta in okorna. Težko je vpeljati morebitne spremembe in bližnjice za izboljšavo uporabniške izkušnje. Prav tako so si vizualno rešitve med seboj zelo podobne. Modeli in transformacije so razširljivi samo tam, kjer so bili predvideni, da so razširljivi. Sledenje spremembam pri modelih je lahko problematično, saj lahko že pri eni sami spremembi modela z uporabo obstoječih orodij pride do večjega števila sprememb po celotnem modelu. Ta problem je še posebej izrazit pri kompleksnih UML modelih.

MDA način razvoja v tem trenutku še ni zaživel v veliki meri. Zato so potrebni razvijalci z višjim nivojem znanja in veščin, ki jih je na trgu manj glede na število klasičnih razvijalcev rešitev.

Konzorcij OMG, ki vodi razvoj MDA, ima slabo zgodovino vpeljave standardov. OMG je standardiziral in vpeljal CORBA standard, ki se ni uspešno vpeljal.

Težko merljiva je vrednost MDA pristopa. Kakor pri tradicionalnem razvoju se tudi MDA pristop ne more izogniti kompleksnosti posameznega problema, ki ga rešuje, čas celotnega razvoja se zato posledično bistveno ne zmanjša. Prav tako ostaja odprto pomembno vprašanje

glede kvalitete transformacije uporabniških vmesnikov.

2.14. Izvršljivi UML

Pri izvršljivem UML gre za metodo in način uporabe modelov brez vmesnega prevoda v izvorno kodo. Takšne modele je mogoče testirati, razhroščevati in prevesti neposredno v izvršilno kodo. Izvršljivi UML je prav tako višje nivojski jezik, kakor so jeziki tretje generacije in je praktična implementacija MDA pristopa.

Sistem je sestavljen iz različnih domen in predstavlja del pogleda domenskega problema.

Domenski modeli je predstavljen iz naslednjih elementov:

Domenski graf

Domenski graf določa splošen pogled na domeno. Primer takšnih pogledov je na primer varnostni aspekt aplikacije, način beleženja delovanja ter relacije z drugimi modeli.

Entitetni model

Nabor entitet, ki so prisotne v domeni. Vsaka entiteta ima lahko določen nabor atributov in operacij ter povezave z drugimi entitetami.

Diagram stanj

Diagram stanj določa prehode med posameznimi stanji entitet v sistemu.

Jezik akcij

Entitetni model in diagram stanj določata statičen pogled na domeno. Jezik akcij je nujno potreben za izvajanje sprememb v entitetnem modelu ali kot posledica prehoda iz posameznega stanja. Trenutno obstaja več različnih jezikov za definiranje akcij.

2.15. Uporaba MDA v podjetjih

Kljub predstavljenim prednostim in nekaterim slabostim se MDA kot primarni način razvoja še vedno ne uporablja zelo pogosto. Podjetja na konferencah predstavljajo, na kakšen način so vpeljala razvoj s pomočjo modelov v svoj interni razvojni proces. Podjetja predstavljajo različne domene, s katerimi se ukvarjajo in s tem povečujejo diverziteto problemov, ki jih takšen način razvoja lahko rešuje [6].

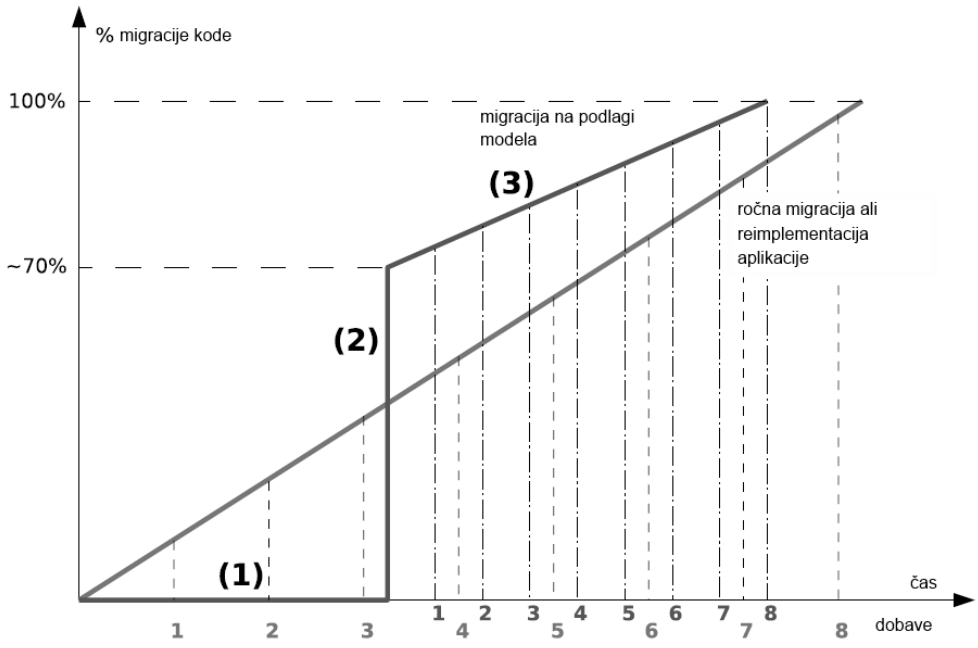
Na spletni strani OMG sestavljajo seznam referenčnih podjetij, ki uporabljajo takšen način razvoja. Prav tako dandanes različna vertikalna področja, kot so zdravstvo, finance, telekomunikacije in podobno, izdelujejo osnovo za izdelavo modelov, neodvisnih od tehničnega okolja. OMG s pomočjo arhitekturne skupine nadzoruje standardizacijo teh domensko specifičnih modelov [7].

Iz dosedanjih izkušenj in na podlagi predstavitev na različnih slovenskih konferencah je praksa v večini manjših in večjih slovenskih podjetjih, da se za razvoj programske opreme še vedno uporablja predvsem klasičen razvoj. Modeli so v praksi uporabljeni večinoma kot podpora analizi problemov. Ko se v praksi pojavi potreba po modelih, so le ti narejeni v UML jeziku, ali pa še vedno velikokrat v nestandardni obliki zaradi potreb po predstavitvi modelov svojim ciljnim strankam, ki standardnih oblik ne znajo brati in jih razumeti. Z uporabo standardnih modelov bi lahko takšna podjetja z vpeljavo že izdelanih ali pa na novo izdelanih transformacij postopoma konvergirala proti arhitekturi s pomočjo modelov. Nekatera predvsem manjša slovenska podjetja se že danes zavedajo prednosti takšnega razvoja, saj jim to prinaša konkurenčno prednost pred večjimi podjetji.

V izbranem primeru gre za migracijo celotnega bančnega okolja, ki ga je izvajal SodiFrance[6], iz stare COBOL in C tehnologije na novejšo tehnologijo JavaEE. Projekt zajema migracijo 42 aplikacij, 800 obrazcev in 990 servisov. Uporabili so pristop migracije na podlagi modelov. Za celoten proces migracije so ocenili, da potrebujejo približno 9000 človek dni. Za pripravo tehnološkega prototipa so porabili 3 mesece ali 2.5 procenta projekta. Po izdelavi tehnološkega prototipa so se lotili izdelave orodij (MIA – Model in Action), potrebnih za migracijo na podlagi ene izbrane aplikacije, za kar so potrebovali nadaljnjih 7 mesecev ali približno 10 procentov projekta. Pri tem so ugotovili, da je za prvi dve fazi zelo težko doseči paralelni razvoj. Orodja, ki so jih izdelali, pa so jim omogočala generacijo približno 70 procentov kode aplikacije. Preostale dele kode, za katere so ugotovili, da je

potrebno ročno inženirsko delo, so označili z ustreznimi »TODO« značkami z referencami na ustrezne točke v analizi. V tretji fazi so se lotili migracije preostalih aplikacij. Pri tem so ugotovili, da lahko za tretjo fazo dodelijo več neodvisnih skupin inženirjev, kjer vsaka skupina dela na svoji aplikaciji.

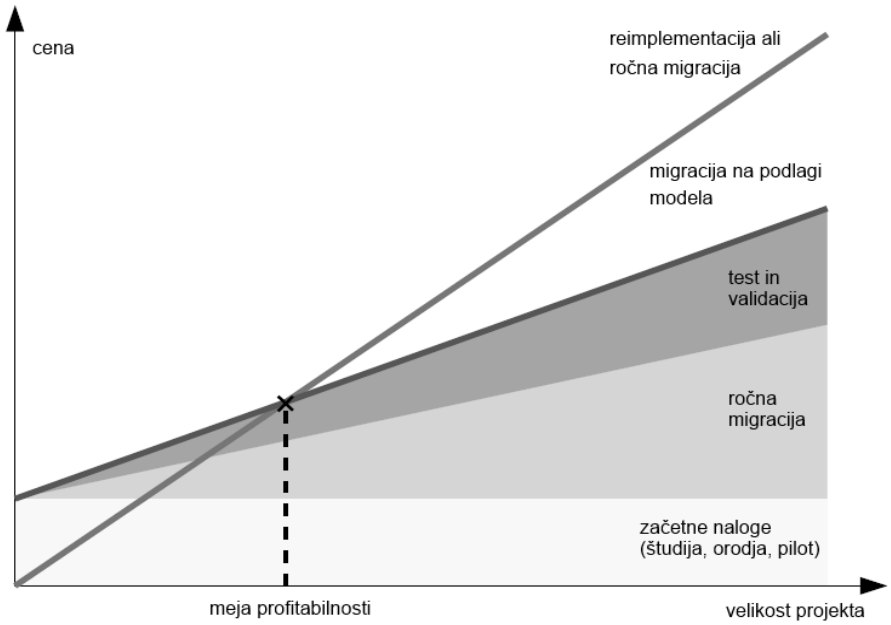
Na sliki 6 lahko vidimo razliko v primerjavi z uporabo tradicionalnega pristopa. V primerjavi s tradicionalnim programskim inženiringom se pri uporabi modelov vidi zamik med začetkom projekta in dostavo prvih verzij strankam. Pri izbranem projektu je bil čas, v katerem je stranka dobila prve dobave aplikacij, 10 mesecev. To je bila ena izmed pomanjkljivosti izbranega pristopa, saj stranka do tega trenutka ni imela vpogleda v napredovanje realizacije projekta. To težavo so poizkušali premostiti na način, da so stranko vključili v sam proces razvoja. Za dobavo preostalih 41 aplikacij so potrebovali 19 mesecev. Vsak mesec so lahko stranki dostavili po dva projekta.



Slika 6: Procent izdelane kode tekom projekta [6]

S pomočjo omenjenega pristopa so ugotovili, da je uporaba razvoja s pomočjo modelov primerna le pri ustrezno velikih projektih, kjer čas izdelave orodij doprinese k skrajšanem času razvoja preostalega dela vsaj za čas izdelave orodij. Na sliki 7 je razvidno, kako se z večanjem velikosti projekta zmanjšuje relativen čas izdelave takšnih orodij. Pri migracijah iz obstoječe v novejšo arhitekturo se ta meja lahko doseže že pri velikosti projekta 1000 človek dni. Za zmanjševanje omenjene meje ima OMG konzorcij ključno vlogo pri standardizaciji

orodij, domenskih modelih in načinu izdelave transformacij. Z uporabo tradicionalnega razvoja so ocenili, da bi bila cena celotnega projekta enkrat višja. Produktivnost posameznega inženirja pa za polovico manjša.



Slika 7: Cena projekta v primerjavi z velikostjo projekta [6]

3. Odprtokodne implementacije

3.1. Xtext

Xtext je orodje, ki omogoča izdelavo domensko specifičnih jezikov in je del projekta Eclipse Modeling. Sama rešitev temelji na knjižnici ANTLR, ki je generator kode na podlagi LL(*) definiranega jezikovnega modela. Orodje omogoča pretvorbo tekstovnega opisa domenskega jezika v Ecore model.

Funkcionalnosti, ki jih Xtext podpira so:

- barvanje sintakse
- navigacija po modelu
- generiranje kode
- primerjava modelov
- avtomatsko dopolnjevanje besedila
- validacija in avtomatično popravljanje napak.

Primer preproste gramatike:

```
grammar org.xtext.Calc with org.eclipse.xtext.xbase.Xbase

generate build "http://www.xtext.org/calc"

Expressions:
	expressions+=Expression*;

Expression:
	Addition;

Addition returns Expression:
	Multiplication (({Plus.left=current} '+' | {Minus.left=current} '-')
right=Multiplication)*;

Multiplication returns Expression:
	PrimaryExpression (({Multi.left=current} '*' | {Div.left=current} '/')
right=PrimaryExpression)*;

PrimaryExpression returns Expression:
	'(' Expression ')' | {NumberLiteral} value=Number;
```

3.2. Acceleo

Acceleo je pragmatična implementacija OMG-jevega MOF Model to Text Language (M2T) standarda. Acceleo je orodje, ki se razvija pod okriljem fundacije Eclipse. Omogoča transformacijo modelov v kodo in je implementacija OMGjevega MOF Model To Text Language (MTL) standarda. Rešitev omogoča integracijo v razvojno okolje Eclipse. Osnova MTL jeziku so MOF modeli, ki so definirani s pomočjo Ecore inačice MOF modela. Jezik MTL določa standarden API za pretvarjanje modelov v tekst. Prednost pred ostalimi orodji je v tem, da so transformacije narejene po definiranem standardu. Trenutno je Acceleo edino in najbolj uporabljeno orodje po standardu MOF M2T. Težave pri uporabi drugih komercialnih rešitev so v tem, da so transformacije nekompatibilne s standardom, zaradi česar je končni uporabnik zaklenjen na njihovo rešitev. Acceleo je zrel produkt in omogoča splošne funkcionalnosti, ki so namenjene hitrejšemu razvoju, kot so barvanje sintakse, detekcija napak, takojšen vpogled v izdelano kodo in sugeriranje pri uporabi podatkov iz modela.

Primer uporabe:

```
[**
 * The documentation of the template generateElement.
 * @param aMongoFile
 */]
[template public generateElement(aMongoFile : MongoFile)]
[comment @main/]
[aMongoFile.generateValidation()/]
[aMongoFile.generateApplicationResources()/]
[aMongoFile.generateMenuConfig()/]
[aMongoFile.generateConverter()/]
[aMongoFile.generateRestService()/]
[for (elem : MongoBean | aMongoFile.eContents(MongoBean))]
[elem.generateDAO()/]
[elem.generateDAOhibernate()/]
[elem.generateModel()/]
[elem.generateManager()/]
[elem.generateManagerImpl()/]
[elem.generateController()/]
[elem.generateViewDetails(aMongoFile)/]
[elem.generateViewList()/]
[elem.generateFlow()/]
[/for]
```

3.3. Eclipse ATL

Eclipse ATL je orodje in jezik za transformacijo modelov v nove modele. ATL je okrajšava za ATLAS transformacijski jezik in omogoča izdelavo transformacij iz več vhodnih modelov v več izhodnih modelov. Eclipse ATL je vgrajen v Eclipse in omogoča standardna razvojna orodja, kot so razhroščevanje in avtomatsko barvanje kode izdelanih transformacij.

Jezik ATL je domenski jezik, sestavljen iz pravil, katera določajo, kako se bo element izvornega modela preslikal v element ciljnega modela. Izvorni modeli se lahko samo berejo, medtem ko se lahko izhodni modeli samo pišejo. Jezik je sestavljen iz deklarativnega in imperativnega dela. Deklarativni del omogoča določanje pravil, s katerimi iščemo vzorce v vhodnem modelu, ki ustrezajo pravilom. Za definiranje pravil se uporablja MDA standard OCL 2.0. Imperativni del omogoča klicanje drugih pravil ter izvajanje akcijskih blokov s sekvenčnimi ukazi, ki določajo, na kakšen način se bo del modela preslikal. Zaradi imperativnega dela je mogoče izdelati transformacije za poljuben izhodni model.

V primeru, ko je potrebno transformirati samo del vhodnega modela bi bilo potrebno izdelati pravila tudi za tiste dele, ki se ne spreminjajo. Takšen način je zamuden, zato Eclipse ATL omogoča poseben način transformacije, kjer se takšna pravila avtomatično zgenerirajo. Takšen način se imenuje »*refining mode*«.

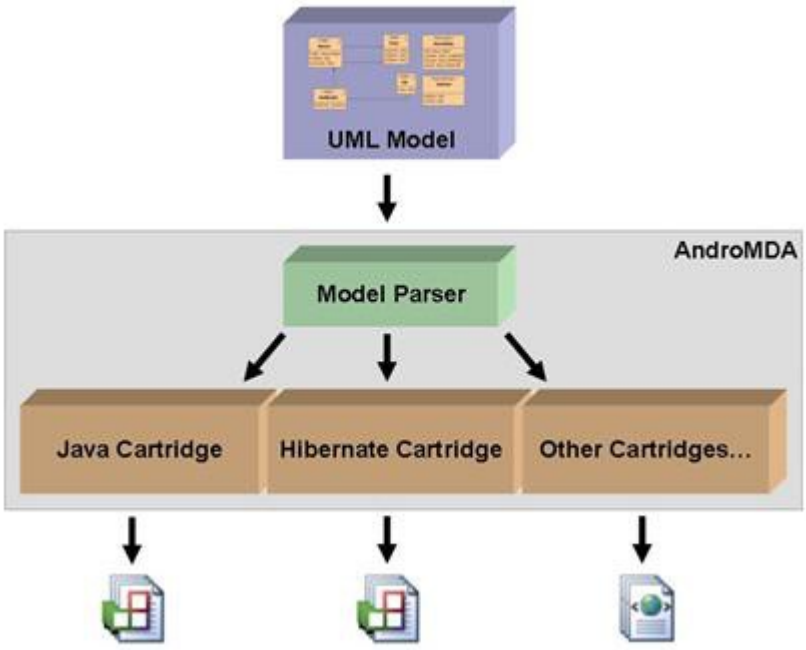
Primer pravila za transformacijo modela MMTree v nov model MMElementList.

```
rule TreeNode2RootElement {
        from
                rt : MMTree!Node
        to
                lstRt : MMElementList!RootElement (
                        name <- rt.name
                )
}
```

3.4. AndroMDA

AndroMDA je orodje za generiranje aplikacij s pomočjo modelov. AndroMDA podpira razširjanje generatorjev kode na nove funkcionalnosti. Generatorji so narejeni s pomočjo Velocity ali FreeMarker knjižnic, namenjenih izdelavi vzorcev, s pomočjo katerih se na podlagi modela izdela izvorna koda. Model, na podlagi katerega rešitev izdela aplikacijo, je UML. Orodje ima že predpripravljene različne transformacije, ki omogočajo generiranje aplikacij v široke naboru tehnologij. Vgrajeni generatorji so narejeni za dostop do baze s pomočjo Hibernate, izdelavo EJB poslovnih beanov, izdelavo Spring poslovne logike, jBPM za izdelavo delovnih tokov, izdelavo spletnih servisov in Struts za izdelavo spletnih aplikacij.

AndroMDA rešitev je zasnovana v celoti modularno (slika 8). Vsak del rešitve je mogoče razširiti ali nadomestiti z lastno implementacijo. AndroMDA podpira različna UML orodja, s katerimi je mogoče izdelati UML modele. Prav tako omogoča tudi uporabo lastnih MOF modelov.



Slika 8: AndroMDA generatorji [10]

Rezultat posamezne transformacije so lahko poljubne tekstovne datoteke. Nabor transformacij

lahko zapakiramo v svoj modul, ki ga lahko uporabimo za generiranje izvorne kode, baznih skript, spletnih strani in podobno.

Slabost uporabljene rešitve je uporaba lastnih oblik transformacij. Zaradi tega je onemogočen preprost prehod na drugo orodje. To predstavlja glaven problem, saj je v zadnjem času razvoj zamrl. Glede na omenjeno modularnost rešitve bi bilo mogoče zamenjati transformacijski jezik v standardnega, vendar bi s tem rešitev izgubila prednost že izdelanih transformacij.

4. Uporaba MDA

4.1. Domenski problem

Za reševanje domenskega problema smo si izbrali zelo splošen problem, ki je prisoten povsod in je dobra osnova za reševanje bolj kompleksnih problemov. Gre za računalniško vodenje in spremljanje procesa beleženja in reševanja napak. Problem bomo razdelil na več neodvisnih problemov, ki se pojavijo pri tradicionalnem razvoju.

Glavna entiteta sistema je naloga, ki vsebuje različne atribute.

Napaka ali naloga lahko vsebuje:

- Projekt v katerem je napaka
- Kratek opis
- Podroben opis
- Kratka oznaka ali šifra
- Verzija v kateri se pojavlja napaka
- Verzija v kateri je rešena napaka
- Kategorija napake
- Resnost napake
- Rešitev napake
- Stanje v katerem se znotraj procesa napaka nahaja.

Ostale entitete so podpora glavni entiteti. Te entitete so različni šifranti. Šifranti so:

- Seznam projektov
- Seznam verzij
- Seznam kategorij
- Resnost napake
- Načini rešitve
- Prioritete napak.

Za vodenje napak smo si zamislili klasičen proces reševanja napak. Možna stanja napake so:

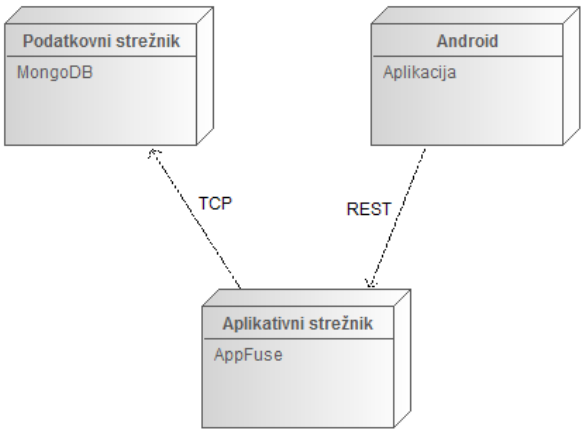
- Nova
- Dodeljena
- Rešena
- Ponovno odprta (neustrezna rešitev)
- Preverjena
- Zaprta.

4.2. Kako smo se lotili problema

Izdelali bomo domenski model v besedilni obliki, ki je razumljiv tudi analitikom, vendar ga je mogoče s pomočjo uporabljenih tehnologij pretvoriti avtomatično v PIM model. Pretvorbo v PIM model bo omogočila integracija med različnimi Eclipse orodji. CIM model smo definirali s pomočjo LL(*) gramatike. Na podlagi gramatike se je definiral PIM meta model v formatu Ecore. Pretvorba iz besedila, ki ustreza definirani gramatiki, smo v PIM model pretvorili s pomočjo knjižnice Xtext. Na podlagi PIM modela smo nato zgenerirali različne PSM modele. Za model PSM smo uporabili specifične oznake v kodi (Annotation), ki jih dandanes uporabljajo tehnologije, uporabljene v rešitvi. PSM model je tako združen kar v kodi s pomočjo teh oznak, ki določajo obnašanje rešitve. Vsaka tehnologija, ki smo jo uporabili, ima svoj nabor oznak. SCXML ima svoj PSM model, ki je v XML obliki in določa diagram poteka.

4.3. Arhitektura rešitve

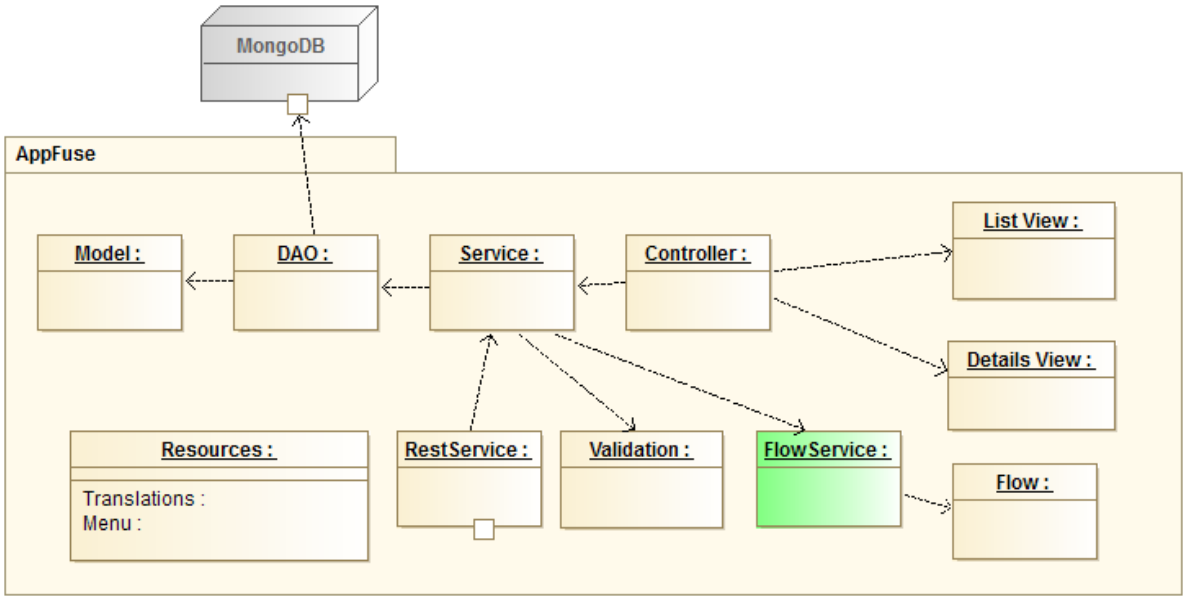
Na najvišjem arhitekturnem nivoju je celotna rešitev sestavljena iz treh delov, kjer se vsak del rešitve izvaja v svojem okolju, kakor je razvidno iz slike 9. Podatkovna baza MongoDB je poganjana na podatkovnem strežniku kot samostojni proces in skrbi za varno shranjevanje podatkov. Za izvajanje aplikacije AppFuse na aplikativnem strežniku skrbi Tomcat aplikativno okolje na JVM izvajalnem okolju. Zaradi uporabe standardnih mehanizmov pri izdelavi aplikacije je mogoče za aplikativni strežnik uporabiti tudi druge aplikativne strežnike, ki ustrezajo JavaEE aplikativnemu okolju. Android aplikacije je ločena rešitev, ki komunicira s pomočjo izdelanih REST spletnih servisov z AppFuse aplikacijo. Android aplikacija se izvaja v Android izvajalnem okolju.



Slika 9: Visoko nivojska arhitektura rešitve

4.3.1. Aplikacija AppFuse

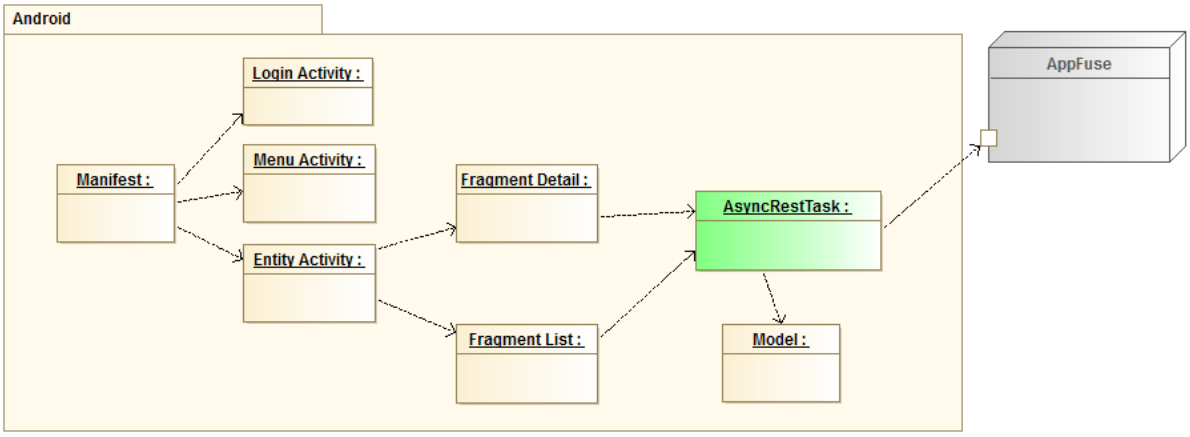
AppFuse rešitev je sestavljena kot trinivojska aplikacija. Kakor je razvidno iz slike 7, je aplikacija sestavljena iz podatkovnega, aplikativnega in prezentacijskega nivoja. Podatkovni nivo je sestavljen iz modela in servisa za dostop do podatkovne baze (DAO). Srednji aplikativni nivo (Service) je zadolžen za izvajanje poslovne logike, med katero sodi validacija in nadzor nad delovnim tokom. Prezentacijska nivoja sta dva. Prvi skrbi za prezentacijo podatkov uporabniku spletne aplikacije in je sestavljen iz kontrolerja in dveh tipov pogledov. Pogleda sta vsebinsko ločena na seznam podatkov in na podrobnosti posameznega podatka. Na sliki je predstavljena arhitektura za eno entiteto. S pomočjo arhitekture na podlagi modela bomo izdelali transformacije, s pomočjo katerih bomo zgenerirali predstavljene elemente aplikacije za vse elemente v modelu. Določene programske elemente bomo izdelali samo enkrat, ker so neodvisni od modela. Takšen je servis za nadzor podatkovnega toka, ki je na sliki 10 označen z zeleno barvo.



Slika 10: Arhitektura AppFuse rešitve

4.3.2. Android aplikacija

Android aplikacije je sestavljena iz različnih med seboj povezanih elementov. Glavni elementi aplikacije, s katerimi se bomo ukvarjali, so predstavljeni na sliki 11. Glavni del aplikacije je manifest, ki določa minimalno verzijo in aktivnosti, ki jih aplikacija vsebuje. Za potrebe reševanja problema smo razdelili aplikacijo v tri aktivnosti, ki skrbijo za prijavo uporabnika, prikazovanje seznama entitet in posamezne entitete. Aktivnost posamezne entitete je sestavljena iz dveh delov, kjer prvi skrbi za prikazovanje seznama entitet, drugi pa skrbi za urejanje podatkov posamezne entitete. Za vsako komponento bomo izdelali transformacijo, ki bo pretvorila posamezno entiteto iz modela v različne komponente aplikacije. Komponenta za asinhrono pridobivanje podatkov, ki je označena z zeleno, ni odvisna od modela, zato za njo ni potrebe po izdelavi transformacije, in bo izdelana na klasičen način.



Slika 11: Arhitektura Android aplikacije

4.4. Praktičen primer uporabe orodij

Za praktičen primer uporabe MDA načina razvoja smo izbrali tehnologije, ki so trenutno najbolj uporabljene pri razvoju novih aplikacij. Tehnologije so bile izbrane neodvisno od tehnologij, ki jih trenutna orodja MDA pokrivajo. Želeli smo ugotoviti težave pri uporabi tehnologij, ki so trenutno pri uporabnikih priljubljene, in na kakšen način lahko takšne tehnologije vpeljemo s pomočjo MDA procesa.

Za reševanje domenskega problema smo izbrali Acceleo in Xtext, ki sta osnovana na odprto kodni rešiti Eclipse Modeling Framework in se razvijata pod okriljem Eclipse Foundation. S pomočjo teh orodij bo prikazana praktična uporaba na primeru ogrodij AppFuse in Androidu.

AppFuse je ogrodje, namenjeno hitremu spletnemu razvoju aplikacij v Javi. Namen ogrodja je predizbor različnih knjižnic v enoten in funkcionalen paket, katerega lahko razvijalec uporabi kot osnovo za razvijanje svojih spletnih aplikacij. AppFuse je sestavljen iz najnovejših in popularnih Java in JavaScript knjižnic. Trenutna verzija povezuje v smiselno celoto naslednje knjižnice:

- Bootstrap in Jquery
- Maven, Hibernate, Spring in Spring Security
- Java 7, Annotations, JSP 2.1, Servlet 3.0
- Spletne knjižnice: JSF, Struts 2, Spring MVC, Tapestry 5, Wicket
- Podpora REST in JAX-WS spletnim servisom
- Podpora Java Persitance API.

AppFuse prinaša ustrezno zasnovo in primer uporabe za naslednje splošne potrebe spletnih aplikacij:

- Avtentikacija in avtorizacija
- Upravljanje z uporabniki
- Prijava, registracija in pošiljanje pozabljenih gesel
- Nalaganje datotek
- Osnovni CRUD
- Podpora različnim razvojnim okoljem (Eclipse, NetBeans in IDEA)
- Podpora različnim aplikativnim strežnikom (Jboss, Tomcat, Jetty, Glassfish)
- Avtomatsko testiranje.

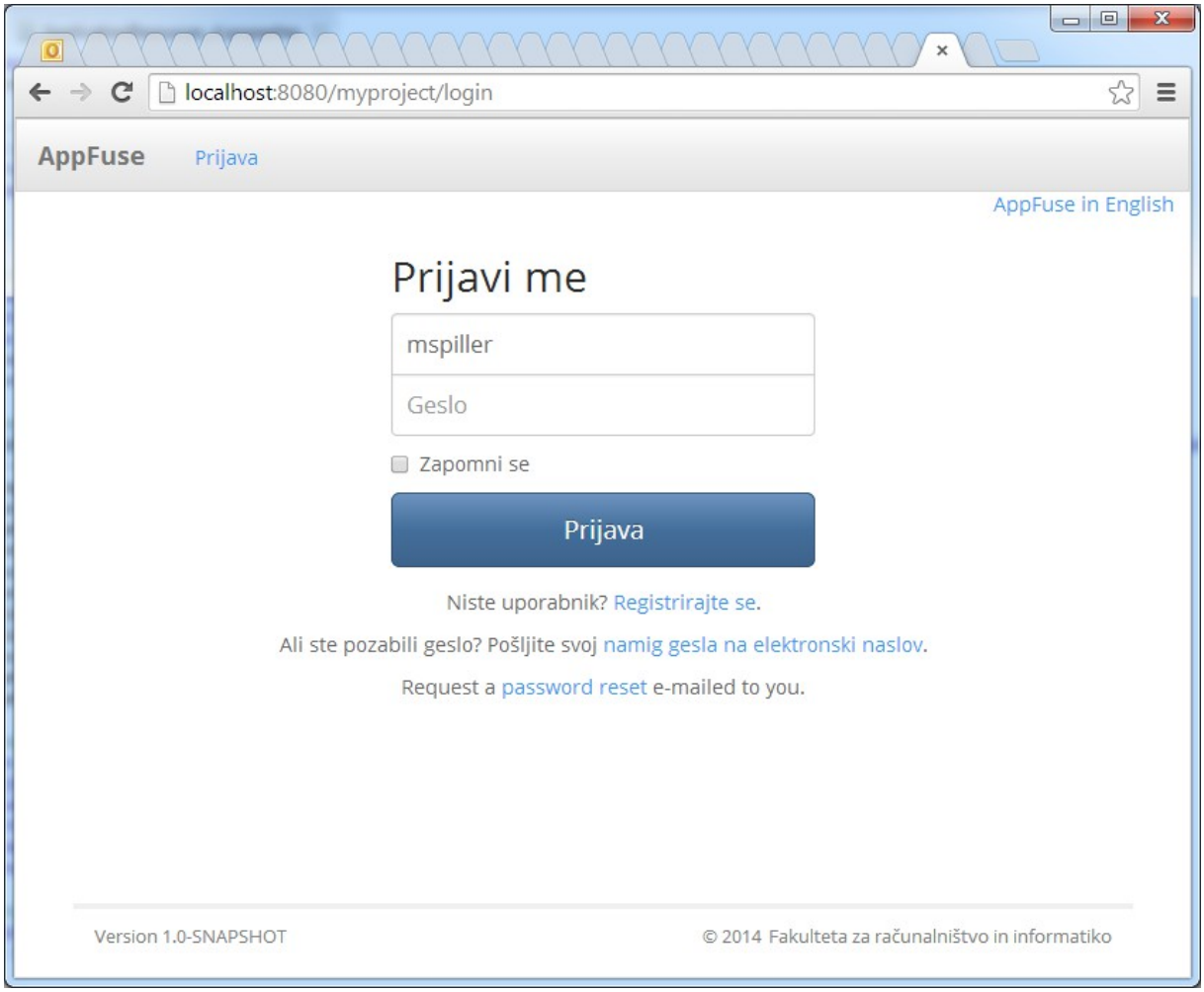
S pomočjo MDA smo želeli narediti uporabno aplikacijo, ki jo lahko končni uporabnik prilagaja samo s pomočjo opisanega modela v obliki, ki je primerna za uporabo analitiku

aplikacij.

Model definira osnovno funkcionalnost aplikacije in se ga lahko kasneje razširi z dodatnimi funkcionalnostmi.

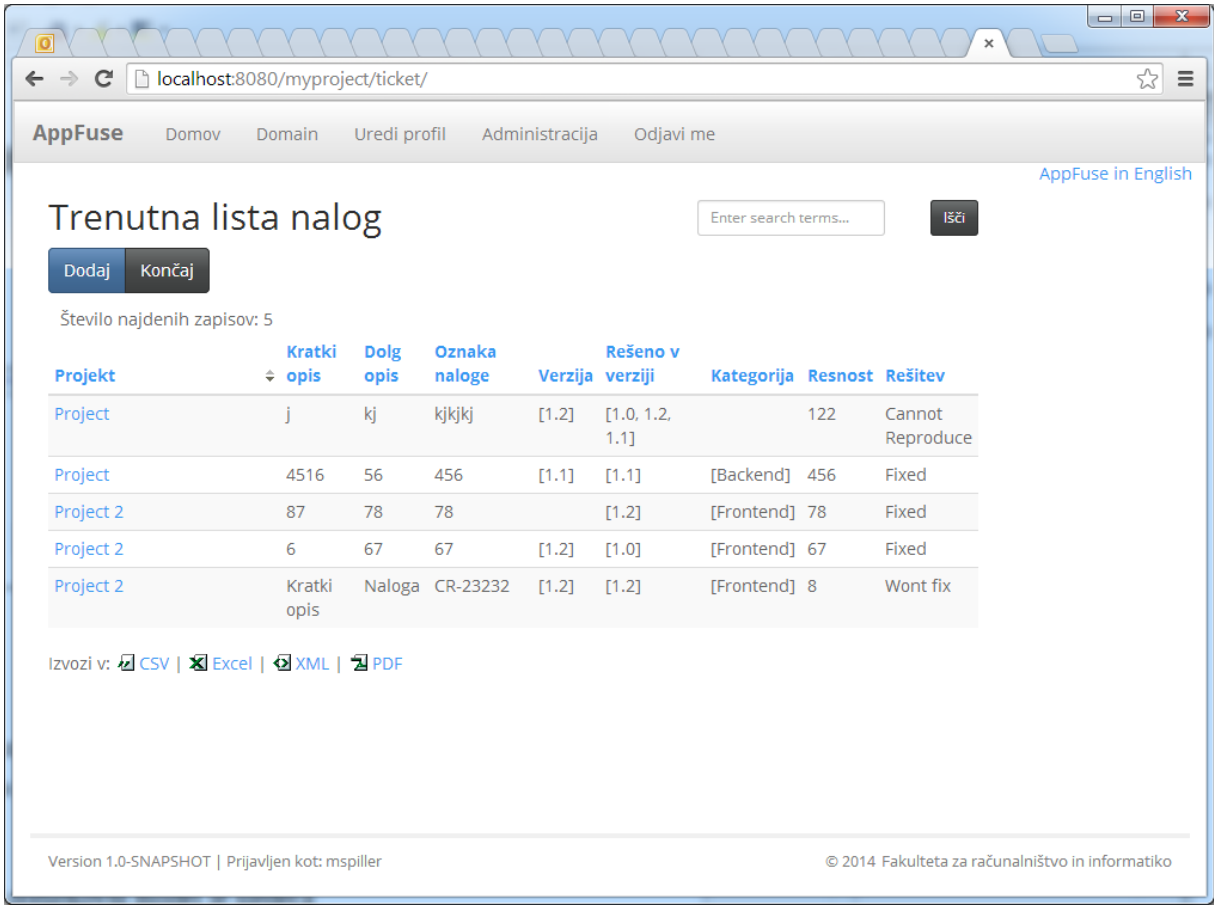
Aplikacije je narejena po MVC principu za izdelavo spletnih aplikacij. MDA generator kode je razdeljen na več modulov, ki skrbijo za generacijo kode na različnih nivojih.

Na sliki 12 je predstavljena vstopna stran do aplikacije. Vstopna stran je že izdelana in vgrajena v AppFuse okolje skupaj z nadzorom pravic in uporabniškimi profili.



Slika 12: Vstopna stran aplikacije

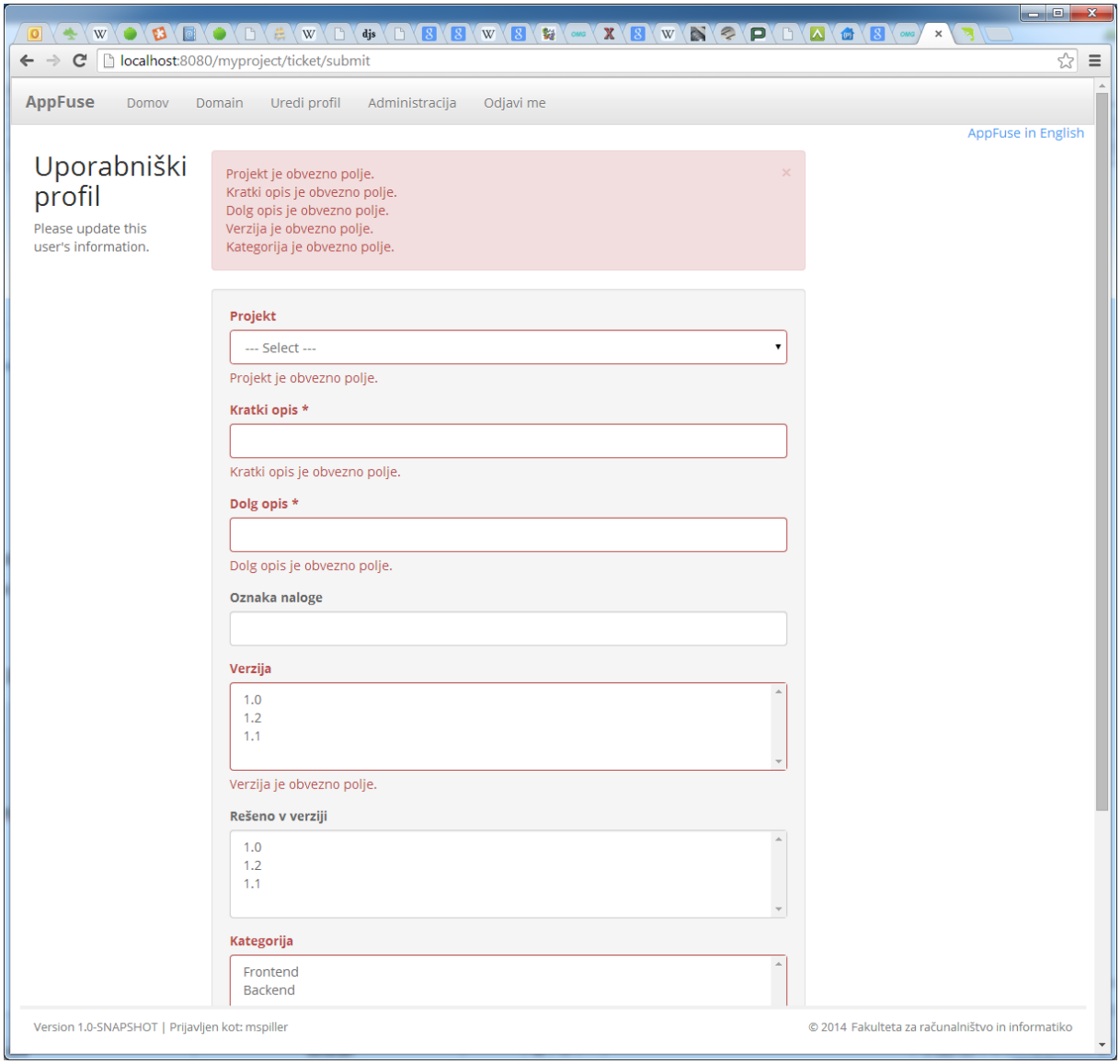
Slika 13 prikazuje zgenerirani del aplikacije, katerega naloga je izpis vseh nalog skupaj s podatki, ki jih posamezna naloga lahko vsebuje.



Projekt	Kratki opis	Dolg opis	Oznaka naloge	Verzija	Rešeno v verziji	Kategorija	Resnost	Rešitev
Project	j	kj	kjkjkj	[1.2]	[1.0, 1.2, 1.1]		122	Cannot Reproduce
Project	4516	56	456	[1.1]	[1.1]	[Backend]	456	Fixed
Project 2	87	78	78		[1.2]	[Frontend]	78	Fixed
Project 2	6	67	67	[1.2]	[1.0]	[Frontend]	67	Fixed
Project 2	Kratki opis	Naloga	CR-23232	[1.2]	[1.2]	[Frontend]	8	Wont fix

Slika 13: Seznam nalog

Dodajanje nove naloge je sestavljeno iz nabora polj, s katerimi lahko uporabnik vnese ali spremeni podatke o posamezni nalogi. Na sliki 14 se prav tako vidi rezultat izdelane komponente za validacijo podatkov o posamezni entiteti.



Slika 14: Vnos nove naloge v aplikaciji

4.4.1. Gramatika jezika

Gramatika jezika, s katerim smo definirali podatkovni model, je sledeča:

```
MongoFile:
       importSection=XImportSection?
       elements+=AbstractElement*;

AbstractElement:
       PackageDeclaration | MongoBean;

PackageDeclaration:
    'package' name=QualifiedName '{' elements+=AbstractElement* '}';

MongoBean:
    name=ValidID '{' features+=AbstractFeature* '}';

AbstractFeature:
   MongoProperty;

MongoProperty:
    (ref?='ref')? (type=JvmTypeReference | inlineType=MongoBean) (many?='*')?
name=ValidID;
```

Jezik vsebuje elemente, operacije in atribute. Sam po sebi je podoben MOF deklaraciji. Xtext omogoča pretvorbo tekstovne vsebine v MOF model. Slika 15 prikazuje pretvorjen model v PIM model.

Slika 15: PIM model aplikacije

4.5. Generacija modela

Za pripravo modela smo predelali ogrodje AppFuse na način, s katerim smo osnovno infrastrukturo spremenili tako, da smo dodali podporo za NoSQL podatkovno bazo. Motivacija za uporabo NoSQL je bila, da lahko na preprost način razširjamo podatkovno strukturo na podlagi modela. Pri RBDMS podatkovni bazi mora biti model podatkovne baze znan vnaprej, medtem ko se NoSQL podatkovna baza prilagaja glede na potrebe modela. Ker MDA način razvoja omogoča izredno dinamično spreminjanje podatkovnega modela, pri prototipnem ali agilnem razvoju manj toga podatkovna baza pohitri razvoj in testiranje različnih modelov ter njihove iterativne dograditve. Za potrebe integracije NoSQL podatkovne baze je bila uporabljena MongoDB podatkovna baza in knjižnica Spring Data, ki je namenjena integraciji nereleacijskih podatkovnih baz v programske rešitve.

Ciljni PSM model je definiran kot JavaBean razred, ki definira podatkovni model kot seznam atributov klasa in metod za dostop do vsebine teh atributov. JavaBean razredi se uporabljajo kot predstavitev podatkovnega modela v Javi. JavaBean razrede se lahko uporabi v ORM knjižnicah, ki preberejo model s pomočjo introspekcije, ga analizirajo in naredijo s pomočjo generatorja kode ovoj »*proxy*«, s pomočjo katerega lahko programer dostopa do podatkov. JavaBean razredi vsebujejo seznam lastnosti modela ter metode za dostop do teh lastnosti. Posamezna lastnost je referenca na drug JavaBean razred ali pa je posamezna lastnost oblike posameznega vgrajenega tipa v Javi. Takšen model je po vsebini zelo podoben MDA MOF modelu.

Za izdelavo takšnega modela smo izdelali transformacijo, ki pretvori PIM model v Javanski PSM model. Model je zaradi neodvisnega načina predstavitve modela uporabljiv v isti obliki v relacijskih in nerelacijskih platformah z uporabo različnih knjižnic. Zaradi tega bi lahko takšen model imenovali tudi PIM model.

Transformacija koda za izdelavo modela je sledeča:

```
[template public generateModel(aMongoBean : MongoBean)]
...

/* Podatkovni model entitete */
public class [aMongoBean.name.toUpperFirst()/] implements Serializable {
    private static final long serialVersionUID = 1L;
    private ObjectId id;

   /* Lastnosti posamezne entitete */
[for (property : MongoProperty | aMongoBean.eContents(MongoProperty))]
[if (property.ref)]
      @DBRef
[/if]
[if (property.many)]
      /* Seznam lastnosti v primeru scenarija “one to many” */
      private List<[property.getType()/]> [property.getName()/]List;
[else]
      private [property.getType()/] [property.getName()/];
[/if]
[/for]

...

}

[/template]


/* pretvorba v ustrezni tip */
[template public getType(param : JvmParameterizedTypeReference)]
[param.type.getType()/]
[/template]

[template public getType(param : JvmGenericType)]
[param.identifier/]
[/template]
```

4.6. Kontroler

Kontroler je sestavni del MVC tehnične rešitve. Odgovoren je za pripravo podatkov in oblikovanje podatkov, primernih za uporabo v pogledu. Za izdelavo kontrolerja pogledov smo izdelali transformacijo, s katero se izdela ustrezni razred, ki služi za manipulacijo in izvajanje akcij s pomočjo pogleda. Kontrolerji so za vsako entiteto iz modela zelo podobni, zato je izvedba modela s pomočjo ene transformacije prihranila čas, potreben za izvedbo kontrolerja.

Transformacijska koda kontrolerja je v ključnih delih prikaza spodaj.

```
[comment encoding = UTF-8 /]
[module controller('/fusemodel/model/generated/FuseModel.ecore',
'http://www.eclipse.org/xtext/common/JavaVMTypes',
'http://www.eclipse.org/xtext/xbase/Xtype')/]

[template public generateController(aMongoBean : MongoBean)]
[file
('src/main/java/com/mycompany/webapp/controller/'.concat(aMongoBean.name).concat('
Controller.java'), false, 'UTF-8')]
/* Autogenerated by controller.mtl */
package com.mycompany.webapp.controller;
...

@Controller
@RequestMapping("/[aMongoBean.name.toLowerFirst()/]")
public class [aMongoBean.name/]Controller extends BaseFormController {
...

    <!-- pridobivanje seznama podatkov iz podatkovne baze -->
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public ModelAndView handleRequest(@RequestParam(required = false, value = "q")
String query) throws Exception {
        Model model = new ExtendedModelMap();
        try {
            model.addAttribute([aMongoBean.name.toUpper()/]_LIST,
[aMongoBean.name.toLowerFirst()/]Manager.search(query, [aMongoBean.name/].class));
        } catch (SearchException se) {
            model.addAttribute("searchError", se.getMessage());
            model.addAttribute([aMongoBean.name.toLowerFirst()/]Manager.getAll());
        }
        return new ModelAndView("[aMongoBean.name.toLowerFirst()/]/list",
model.asMap());
    }

...
    <!-- pridobivanje posameznega podatka iz podatkovne baze -->
    @ModelAttribute("[aMongoBean.name.toLowerFirst()/]")
    protected [aMongoBean.name/] load[aMongoBean.name/](final HttpServletRequest
request) {
        final String [aMongoBean.name.toLowerFirst()/]Id =
request.getParameter("id");
        if (/*isFormSubmission(request) &&*/
StringUtils.isNotBlank([aMongoBean.name.toLowerFirst()/]Id)) {
            return
[aMongoBean.name.toLowerFirst()/]Manager.get([aMongoBean.name.toLowerFirst()/]Id);
        }
        return new [aMongoBean.name/]();
    }
```

4.7. Izdelava pogleda

Pogled je izdelan na podlagi JSP standarda. Pogled je zadolžen za pridobivanje informacij iz modela in za prikazovanje le teh uporabniku. Za potrebe aplikacije smo morali izdelati dva pogleda za vsako entiteto. Vsak pogled je specifičen atributom, ki jih posamezna entiteta vsebuje. Izdelali smo dve transformaciji.

Prva transformacija prikazuje tabelo s podatki, ki so predstavljeni v obliki posamezne entitete. Za prikazovanje tabele smo uporabil Apache DisplayTag knjižnico, ki ima že definiran svoj model v obliki JSP značk za prikazovanje tabelaričnih podatkov, zato je to postal naš PSM model za prikazovanje podatkov. Knjižnica definira značke v XML obliki, ki jih je mogoče vključiti v JSP stran.

Glavni del pogleda je sestavljen iz tabele, ki vsebuje vse lastnosti entitete, ter povezavo na urejanje podatkov instance entitete.

```
    <display:table name="[aMongoBean.name.toLowerFirst()/]List" cellspacing="0"
cellpadding="0" requestURI=""
                   defaultsort="1" id="users" pagesize="25" class="table table-
condensed table-striped table-hover" export="true">

[for (property : MongoProperty | aMongoBean.eContents(MongoProperty))]
[if (aMongoBean.eContents(MongoProperty)->first() = property)]
        <!-- posamezna kolona v tabeli z vrednostjo iz modela -->
        <display:column property="[property.name.toLowerFirst()/]"
escapeXml="true" sortable="true" titleKey="[aMongoBean.name.toLowerFirst()/].
[property.name.toLowerFirst()/]" style="width: 25%"
                        url="/[aMongoBean.name.toLowerFirst()/]/edit?from=list"
paramId="id" paramProperty="id"/>
...
[/if]
[/for]
        <!-- akcije za izvoz entitet v excel, csv in PDF obliko -->
        <display:setProperty name="export.excel.filename"
value="[aMongoBean.name/] List.xls"/>
        <display:setProperty name="export.csv.filename" value="[aMongoBean.name/]
List.csv"/>
        <display:setProperty name="export.pdf.filename" value="[aMongoBean.name/]
List.pdf"/>

    </display:table>
```

Naslednja transformacija služi urejanju podatkov entitete.

```
    <form:form commandName="[aMongoBean.name.toLowerFirst()/]" method="post"
action="submit" id="userForm" autocomplete="off"
                cssClass="well" onsubmit="return validateUser(this)">
        <!-- skrito polje za določanje unikatne oznake instance entitete -->
        <form:hidden path="id"/>
<%--         <form:hidden path="version"/> --%>
        <input type="hidden" name="from" value="<c:out value="${param.from}"/>"/>

        <!-- seznam urejevalnikov za posamezno lastnost entitete -->
[for (property : MongoProperty | aMongoBean.eContents(MongoProperty))]
[property.type.getFormEditor(property, aMongoBean, aMongoFile)/]
[/for]
    </form:form>
```

Del naslednje transformacijske kode služi za izdelavo besedilnega ali številčnega vnosnega polja.

```
[if (param.getIdentifier() = 'java.lang.String' or param.getIdentifier() =
'java.lang.Integer')]
        <spring:bind path="[aMongoBean.name.toLowerFirst()/].[property.name/]">
        <div class="form-group${(not empty status.errorMessage) ? ' has-error' :
''}">
        </spring:bind>
            <appfuse:label styleClass="control-label"
key="[aMongoBean.name.toLowerFirst()/].[property.name/]"/>
            <form:input cssClass="form-control" path="[property.name/]"
id="[property.name/]"/>
            <form:errors path="[property.name/]" cssClass="help-block"/>
        </div>
```

4.8. Spletni servisi REST

Spletni servisi so sestavni del vsake bolj kompleksne rešitve. Za potrebe integracije z mobilnimi napravami smo povezali podatkovni model s pomočjo REST spletnih servisov. Za implementacijo smo uporabili Apache CXF knjižnico. Apache CXF je odprtokodna knjižnica, ki omogoča izdelavo spletnih servisov na podlagi JAX-WS in JAX-RS Java standardov. Knjižnica omogoča različne načine povezovanja. Med drugim omogoča povezavo preko HTTP, CORBA, JMS (Java Message Queue). JAX-RS ima definiran svoj model, ki ga je mogoče specificirati s pomočjo anotacij.

Osnova za izdelavo spletnih servisov je nadzorni servis, ki skrbi za upravljanje s podatki ter za delovne tokove. Za definiranje spletnega servisa smo uporabili s pomočjo anotacij izdelan vmesnik RestService, ki ima definirana osnovna mapiranja ukazov na servisnem nivoju. Ti ukazi so standardni za pridobivanje, vnašanje in spreminjanje podatkov in uporabljajo klasične HTTP GET, POST in DELETE metode.

Koda transformacija za generiranje spletnega servisa:

```
[template public generateManagerImpl(aMongoBean : MongoBean)]
[file
('src/main/java/com/mycompany/service/impl/'.concat(aMongoBean.name).concat('Manag
erImpl.java'), false, 'UTF-8')]

/**
 * Implementation of [aMongoBean.name/]Manager interface.
 */
@Service("[aMongoBean.name.toLowerFirst()/]Manager")
@Path("/[aMongoBean.name.toLowerFirst()/]")
public class [aMongoBean.name/]ManagerImpl extends
GenericManagerImpl<[aMongoBean.name/], String> implements
[aMongoBean.name/]Manager {
    [aMongoBean.name/]Dao [aMongoBean.name.toLowerFirst()/]Dao;

    @Autowired
    /* konstruktor za inicializacijo DAO dostopa do podatkovne baze */
    public [aMongoBean.name/]ManagerImpl([aMongoBean.name/]Dao
[aMongoBean.name.toLowerFirst()/]Dao) {
        super([aMongoBean.name.toLowerFirst()/]Dao);
        this.[aMongoBean.name.toLowerFirst()/]Dao =
[aMongoBean.name.toLowerFirst()/]Dao;
    }
}
[/file]
[/template]
```

4.9. Delovni tok

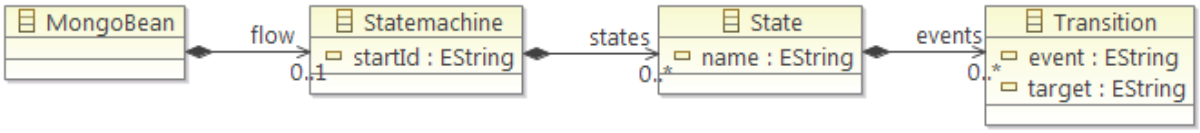
Delovni tok smo definirali s pomočjo SCXML W3C standarda. Standard definira XML format modela, s katerim definiramo možna stanja, v katerih se entiteta lahko nahaja, kot tudi prehode med definiranimi stanji. Za izvajanje zgenerirane definicije delovnega toka smo uporabili Apache knjižnico Commons SCXML, ki prebere definiran SCXML format in ga prevede v izvršljivo kodo.

Izdelali smo PIM model (slika 16), s katerim smo definirali možne prehode stanj za posamezno entiteto. Vsaka entiteta lahko vsebuje svoj delovni tok. Definicijo delovnega toka smo definirali tako, da smo razširili osnovni domenski model v naslednji iteraciji razvoja aplikacije, in rešitvi dodali dodatno funkcionalnost.

Delovni tok smo definirali v domenskem jeziku na sledeči način. Definirali smo nov element Statemachine, ki lahko vsebuje seznam stanj. Dodatno takšen element vsebuje tudi začetno stanje entitete, kateri se določi pri kreiranju nove instance entitete. Vsako stanje vsebuje seznam dogodkov, s katerimi sprožimo prehod iz trenutnega v novo ciljno stanje.

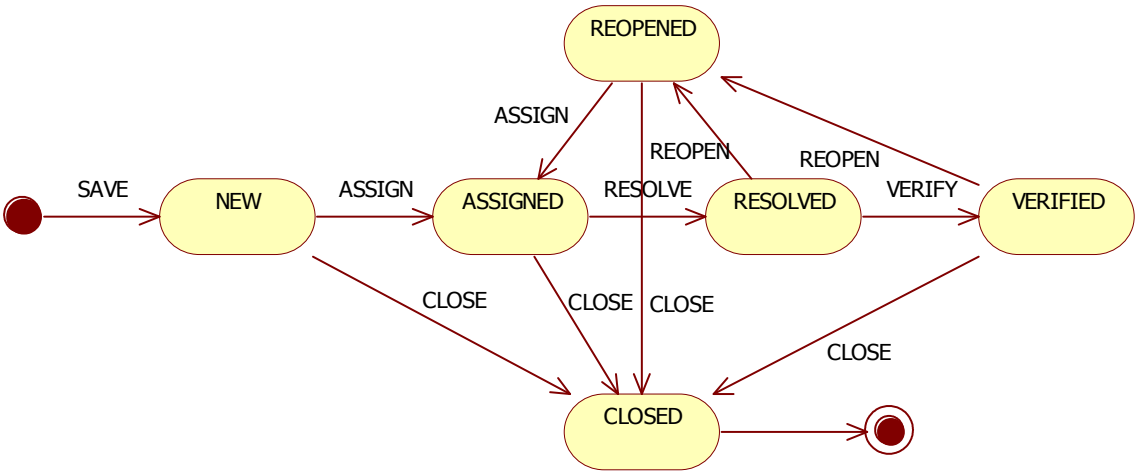
Realizacija domenskega jezika:

```
Statemachine:
	'flow' '{' 'start' startId=ID states+=State* '}' ;

State:
	'state' name=ID '{' events+=Transition* '}' ;

Transition:
	event=ID '=>' target=ID ;
```



Slika 16: Vizualizacija definicije PIM modela

Kakor je prikazano na sliki 17 smo definirali delovni tok za entiteto »Ticket«, ki skrbi za stanje, v katerem se lahko določen zahtevek nahaja, ter prehode med posameznimi stanji.



Slika 17: Diagram delovnega toka

Delovni tok smo nato pretvorili v predhodno definiranem domenskem jeziku.

```
Ticket {
	Attributi entitete ...
	flow {
		start CREATE
		state CREATE {
			SAVE => NEW
		}
		state NEW {
			ASSIGN => ASSIGNED
			CLOSE => CLOSED
		}
		state ASSIGNED {
			RESOLVE => RESOLVED
			CLOSE => CLOSED
		}

		...

		state VERIFIED {
			CLOSE => CLOSED
		}
		state CLOSED { }
	}
}
```

Iz formalnega besedilnega opisa smo izdelali s pomočjo Xtext rešitve na podoben način kakor prej nov objektni model PIM. Nato smo s pomočjo MTL standarda izdelali transformacijo za pretvorbo PIM v PSM model. PSM model je definiran po standardu SCXML. Transformacija za izdelavo tega modela se sprehodi preko vseh entitet, ki imajo definiran delovni tok, in ga izdela na podlagi PIM modela:

```
[template public generateFlow(aMongoBean : MongoBean)]
[if (aMongoBean.flow.oclIsUndefined() <> true )]
[file ('src/main/resources/flows/'.concat(aMongoBean.name).concat('.xml'), false,
'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
initial="[aMongoBean.flow.startId/]">
[for (state : State | aMongoBean.flow.eContents(State))]
    /* posamezno stanje v katerem se entiteta lahko nahaja */
    <state id="[state.name/]">
[for (transition : Transition | state.eContents(Transition))]
        /* možni prehodi v novo stanje na podlagi dogodka */
        <transition event="[transition.event/]" target="[transition.target/]"/>
[/for]
    </state>
[/for]
</scxml>
[/file]
[/if]
[/template]
```

Za uporabo delovnega toka smo izdelali pomožni klas FlowUtil, ki skrbi za pridobivanje podatkov iz zgeneriranih delovnih tokov. Glavni namen tega razreda je nalaganje SCXML datoteke in pridobivanje informacij o možnih prehodih stanj glede na trenutno stanje entitete. Glede na to, da pomožni razred lahko operira z vsemi entitetami naenkrat in ni odvisen od modela posamezne entitete, ga ni bilo smiselno izdelati s pomočjo MDA načina razvoja.

V transformacijo pogleda urejanja transformacije smo dodali seznam akcij, ki jih lahko uporabnik izvede. Seznam akcij je sestavljen iz akcijskih gumbov, kot je razvidno spodaj.

```
[if (aMongoBean.flow.oclIsUndefined() <> true )]
         <!-- seznam gumbov, ki odražajo akcije delovnega toka -->
         <c:forEach var="a" items="${actions}">
            <button type="submit" class="btn btn-primary" name="action" value="$
{a}" onclick="bCancel=false">
               <i class="icon-ok icon-white"></i> <fmt:message
key="button.action.${a}"/>
            </button>
         </c:forEach>
[else]
...
[/if]
```

Atribut MVC modela ${actions} smo napolnili z rezultatom pomožnega klasa, ki vrne seznam vseh možnih akcij glede na trenutno izbrano instanco entitete. Transformacijo za izdelavo kontrolerja smo razširili v metodi, ki je zadolžena za izdelavo modela za prikaz podatkov pri urejanju in dodajanju nove entitete, in ji dodali klic na statično metodo FlowUtil.getActions. Uporaba statične metode v transformaciji je razvidna spodaj.

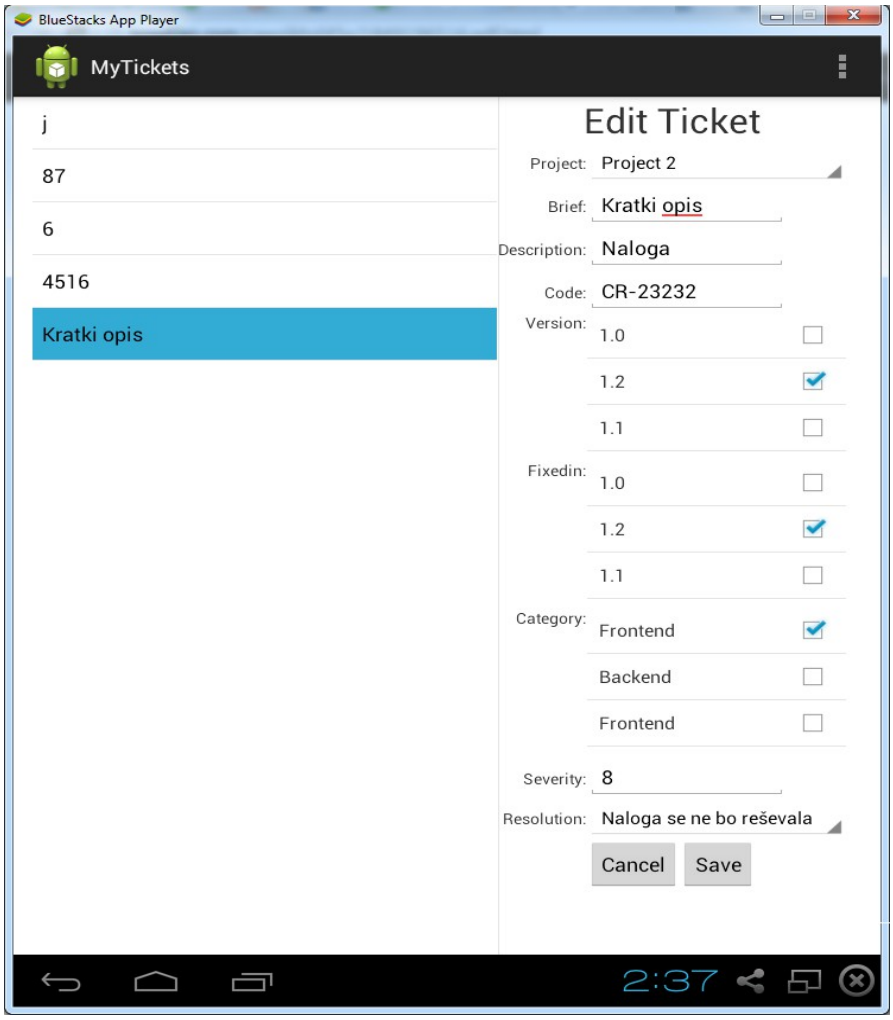
```
@RequestMapping(value = { "/new", "/edit" }, method = RequestMethod.GET)
...
[if (aMongoBean.flow.oclIsUndefined() <> true )]
      /* atribut MVC modela s katerimi se določi seznam akcij */
      model.addAttribute("actions", FlowUtil.getActions([aMongoBean.name/].class,
[aMongoBean.name.toLowerFirst()/].getStatus()));
[/if]
...
```

4.10. Android aplikacija

Android aplikacija se povezuje s prej opisanimi spletnimi servisi. Za mobilne naprave smo pripravili nov podatkovni model, ki je primeren za komunikacijo s pomočjo REST protokola in je prilagojen mobilnim napravam. Aplikacija uporablja podobne koncepte kakor jih spletna aplikacija, tako da smo za osnovo vzeli vse do sedaj izdelane transformacije, in jih prilagodili na drugo izvajalno okolje. Glede na specifičnost platforme niso bile vse transformacije konceptualno uporabne. Zato je bilo potrebno izdelati nekatere nove transformacije. Izziva smo se lotili na način, da smo uporabili AppFuse transformacijo, in ji spremenili posamezne dele glede na drugačen PSM model. Glede na to, da sta si platformi zelo različni, smo morali nekatere transformacije narediti od začetka. Takšni transformaciji sta bili med drugim transformaciji za izdelavo aktivnosti in manifest android aplikacije. Glede na to, da smo z izdelavo transformacij v spletni aplikaciji že pridobili nekaj izkušenj, je bil glavni izziv, kako tehnično izdelati Android aplikacijo, in ne, kako prilagoditi transformacije, ki smo jih imeli že pripravljene za izdelani model.

Na začetku reševanja problema smo si zadali, da izdelamo aplikacijo, ki omogoča uporabniku samo vnašanje posamezne naloge preko mobilnega telefona (slika 18). Tekom razvoja smo prišli do spoznanja, da je čas izdelave aplikacije omejenega dela modela skoraj isti, kakor je čas za izdelavo aplikacije, ki vsebuje celotno funkcionalnost.

Izdelavo oziroma prilagoditve transformacije smo se lotili na način, da smo izdelali novo Android aplikacijo, za katero smo izdelali koncept na podlagi ene entitete. Izdelali smo ustrezne aktivnosti in poglede ter dopolnili definicije, ki so potrebne za poganjanje aplikacije. Pri izdelavi smo si zapisovali in označili vse dele, katere smo želeli izdelati s pomočjo modela. Vse te dele smo nato uporabili pri prilagoditvah obstoječih transformacij ali pri izdelavi novih transformacij. Nekatere dele kode smo izdelali neodvisno od modela, zato za njih ni bilo potrebno izdelati dodatne transformacije. Takšen del so bili asinhroni razredi za klice REST servisov, ki smo jih izdelali s pomočjo generičnih razredov. Vsaka instanca razreda je sprejela kot parameter konstruktorja razred REST modela, s pomočjo katerega smo lahko izdelali JSON predstavitev podatkov. Izdelano JSON predstavitev smo nato preko vgrajene funkcionalnosti za komunikacijo preko spleta uporabili za pošiljanje podatkov v spletno aplikacijo. Pri generaciji modela smo nato izdelali samo končno uporabo teh razredov.



Slika 18: Urejanje naloge v Android aplikaciji

4.10.1. Model spletnih servisov

Modeli spletnih servisov so navadni POJO »*plain java object*« razredi. Koda transformacije je bazirana na transformaciji iz modela AppFuse aplikacije in je prikazana spodaj.

```
[template public generateAndroidModel(aMongoBean : MongoBean)]
[file ('src/org/fri/sampleapp/model/'.concat(aMongoBean.name).concat('.java'),
false, 'UTF-8')]

/* Autogenerated by model.mtl */
package org.fri.sampleapp.model;

/* Podatkovni model entitete */
public class [aMongoBean.name.toUpperFirst()/] implements Serializable {

    private static final long serialVersionUID = 1L;
    private String id;

    /* Lastnosti posamezne entitete */
[for (property : MongoProperty | aMongoBean.eContents(MongoProperty))]
[if (property.many)]
      /* Seznam lastnosti v primeru scenarija “one to many” */
      private List<[property.getType()/]> [property.getName()/]List;
[else]
      private [property.getType()/] [property.getName()/];
[/if]
[/for]

...

[/file]
[/template]
```

4.10.2. Asinhroni klici servisov

Za pravilno uporabo uporabniškega vmesnika je pomembno, da vsi daljši klici ne blokirajo niti uporabniškega vmesnika. Takšne klice je potrebno zapakirati v posebne klase, ki skrbijo za asinhrono izvajanje le teh. Takšen klas se imenuje v Androidu AsyncTask in smo ga uporabili za klicanje REST spletnih servisov. Med izvajanjem asinhrone naloge se prikazuje napredek s pomočjo komponente uporabniškega vmesnika, ki opravlja funkcijo črte napredka. Med izvajanjem asinhrone naloge je mogoče pošiljati posodobitve statusa izvajalni niti uporabniškega vmesnika. Implementacija asinhrone niti je narejena splošno na način, ki omogoča pridobivanje podatkov neodvisno od tipa podatkov. V ta namen smo uporabili

posplošene razrede v javi. Ker gre za posplošen razred, kjer se model uporablja samo kot vhodni tip, uporaba MDA pristopa ni primerna, ker implementacija razreda ni odvisna od posamezne instance modela.

4.10.3. Android aktivnost in fragmenti

Android aplikacija je sestavljena iz nabora aktivnosti. Vsaka posamezna aktivnost vsebuje logiko, ki se ukvarja z enim posameznim zaslonom aplikacije. Aktivnost je lahko sestavljena iz več kot enega fragmenta, ki posamezen zaslon deli na logične enote. Aktivnost lahko glede na velikost naprave prikaže različno število fragmentov naenkrat. Fragment je najmanjša samostojna zaključena celota, ki se ukvarja s predstavitvijo in manipulacijo vsebine. V fragmentu smo povezali komponente, ki predstavljajo podatke v pogledu z akcijami, ki se bodo zgodile, ko jih bo uporabnik sprožil. Fragment smo izdelali s pomočjo transformacije, ki pretvori model v kodo, s katero se shranijo podatki v REST servis.

Primer glavne akcije na fragment za urejanje podatkov entitete je gumb shrani, za katerega smo izdelali transformacijo v naslednji obliki:

```
rootView.findViewById(R.id.
[aMongoBean.name.toLowerFirst()/]_save).setOnClickListener(
    new View.OnClickListener() {
       @Override
       public void onClick(View view) {
       //seznam lastnosti entitete, s pomočjo katere se nastavijo lastnosti
[for (property : MongoProperty | aMongoBean.eContents(MongoProperty))]
[property.type.getFormEditorSave(property, aMongoBean, aMongoFile)/]
[/for]

       //REST servis za pošiljanje spremenjenih podatkov na strežnik
       new UploadAsyncTask(getActivity(), callback, mItem,
[aMongoBean.name/].class).execute();
   }
});
```

4.10.4. Android pogledi

Android aplikacija je sestavljena iz pogledov, ki so XML datoteke. Pogled je tipično sestavljen iz komponent, ki so sestavni del Androida. Nekatere komponente lahko vsebujejo druge komponente in so namenjene sestavljanju uporabniškega vmesnika posamezne aktivnosti.

Osnovni gradnik posameznega pogleda je GridLayout, ki skrbi za postavljanje komponent v tabelarično obliko. V tabeli je v prvi koloni labela polja, v drugi koloni se nahaja vnosno polje. Prikaz polja je odvisen od tipa posamezne lastnosti v modelu. Na koncu pogleda se nahajata gumba Shrani in Prekliči.

Transformacijska koda, ki je prikazana spodaj, pretvarja model v urejevalnik posamezne entitete.

```
    <GridLayout
        ... >
        <!-- labela z imenom entitete -->
        <TextView
            android:layout_columnSpan="3"
            android:layout_gravity="center_horizontal"
            android:text="Edit [aMongoBean.name/]"
            android:textSize="32dip" />
        <!-- seznam urejevalnikov posameznih lastnosti entitete -->
[for (property : MongoProperty | aMongoBean.eContents(MongoProperty))]
[property.type.getFormEditor(property, aMongoBean, aMongoFile)/]
[/for]
        <Space
            android:layout_gravity="fill" />

        <!-- gumba za preklic in shranjevanje sprememb -->
        <Button android:id="@+id/[aMongoBean.name.toLowerFirst()/]_cancel"
android:text="Cancel" />
        <Button android:id="@+id/[aMongoBean.name.toLowerFirst()/]_save"
android:text="Save" />
    </GridLayout>
```

Vsaka lastnost modela ima glede na tip narejen svoj del transformacije, ki se imenuje getFormEditor. Transformacija GetFormEditor, ki je prikazana spodaj, transformira model v labelo in vnosno polje.

```
[template public getFormEditor(param : JvmParameterizedTypeReference, property:
MongoProperty, aMongoBean: MongoBean, aMongoFile: MongoFile)]
[if (param.getIdentifier() = 'java.lang.String')]
        <!-- labela z imenom lastnosti entitete -->
        <TextView
            android:layout_width="wrap_content"
            android:layout_gravity="right"
            android:text="[property.name.toUpperFirst()/]:" />

        <!-- urejevalnik lastnosti entitete -->
        <EditText
            android:id="@+id/
[aMongoBean.name.toLowerFirst()/]_[property.name.toLowerFirst()/]"
            android:layout_columnSpan="2"
            android:ems="8"
            android:inputType="text" />
...
```

4.10.5. Android manifest

Da je Android aplikacija popolna, je potrebno določiti manifest aplikacije. Manifest je zapisan v XML zapisu, katerega oblika je definirana z ustrezno shemo. Manifest vključuje ime in verzijo aplikacije, potrebne pravice in minimalno verzijo Androida, na katerem bo aplikacija izvajana. Prav tako vsebuje informacije o vseh aktivnosti, ki bodo v aplikaciji uporabljene. Za izdelavo manifesta smo zato izdelal transformacijo, s katero se iz seznama entitet izdela za vsako entiteto dve definiciji aktivnosti. Prva aktivnosti služi prikazovanju seznama entitet, druga aktivnost se uporablja za prikazovanje in urejanje podatkov posamezne entitete CIM modela. Koda transformacije je prikazana spodaj.

```
[for (elem : MongoBean | aMongoFile.eContents(MongoBean))]
        <!-- aktivnost, ki skrbi za seznam entitet -->
        <activity
            android:name="org.fri.sampleapp.[elem.name/]ListActivity"
            android:label="@string/app_name" >
        </activity>

        <!-- aktivnost, ki skrbi za urejanje entitete -->
        <activity
            android:name="org.fri.sampleapp.[elem.name/]DetailActivity"
            android:label="@string/title_[elem.name.toLowerFirst()/]_detail"
            android:parentActivityName=".[elem.name/]ListActivity" >
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".[elem.name/]ListActivity" />
        </activity>
[/for]
```

5. Sklepne ugotovitve

V praktičnem delu smo odkrili, da smo za izdelavo arhitekturnega prototipa porabili približno 30 procentov časa. Arhitekturni prototip je zahteval več časa, ker je bil sestavljen iz dveh tehnološko precej različnih aplikacij. Za izdelavo domenskega modela smo potrebovali samo 10 procentov časa. Izdelava vseh potrebnih transformacij pa je trajala kar 50 procentov časa. Preostalih 10 procentov je bilo namenjeno testiranju. Glede na dosedanje izkušnje bi porabili za podobno aplikacijo s pomočjo tradicionalnega razvoja le približno eno tretjino časa. To pomeni, da se takšen razvoj pri podobnem projektu časovno izplača šele pri vsaj trikrat večjemu projektu. Ocenjujemo, da bi se v primeru, ko bi bile na voljo kvalitetne transformacije, ki bi že upoštevale želeno arhitekturo in tehnologijo, lahko skrajšal čas razvoja v našem primeru na približno tretjino, in bi se izenačil s klasičnim razvojem tudi pri manjših projektih.

Ali in pod kakšnimi pogoji programiranje v jeziku višjem od tretje generacije prinaša višjo hitrost razvoja, manj napak in hitrost, ostaja v tem trenutku še odprto vprašanje. Na študijskem primeru [9] je bil razvoj hitrejši za približno 30 procentov, prav tako je bilo manj napak. MDA način razvoja prinaša kar nekaj prednosti glede na klasičen razvoj, vendar do danes pristop še vedno ni v pretiranem razmahu. Glavna težava je v številu orodij in kvaliteti le teh. Orodja vsebujejo zelo omejeno število pred pripravljenih transformacij ter definicije modelov, ki bi pokrivale trenutne potrebe v računalniški industriji. Modeli in transformacije, ki že obstajajo, so v tem trenutku že zastareli, saj uporabljajo tehnologije, ki niso več v uporabi (CORBA, EJB 2.x, in tako dalje). Težava za to leži v tem, da orodja ne dohajajo tehnologij, ki se neprestano posodabljajo in pridobivajo nove in nove funkcionalnosti. Razlog leži v premajhnem interesu glede na potrebe, ker je v ospredju še vedno klasičen razvoj. Podjetja, ki uporabljajo MDA pristope, ne objavljajo orodij, izdelanih za svoje potrebe, saj jim le ta prinašajo večjo konkurenčno prednost. Zaradi tega je trenutno investicija v MDA način razvoja večja in je posledično uporabljiva le pri večjih projektih, pri katerih je veliko kode, ki je zgrajena na podlagi več ponavljajočih se vzorcev. Pri veliki količini ponavljajočih se vzorcev, ki se pojavljajo v rešitvi, MDA način razvoja odtehta čas, ki je potreben za izdelavo in pripravo definicij modelov in transformacij.

6. Literatura

[1] A. Kleppe, J. Warmer, W. Bast, MDA Explained: The Model Driven Architecture™: Practice and Promise, Addison Wesley, 2003

[2] S. J. Mellor, K. Scott, A. Uhl, D. Weise, MDA Distilled: Principles of Model-Driven Architecture, Addison Wesley, 2004

[3] D. Milicev, Model-Driven Development with Executable UML, Wiley Publishing, Inc, 2009

[4] D. S. Frankel, Model Driven Architecture Applying MDA to Enterprise Computing, Wiley Publishing, 2003

[5] O. Pastor, J. C. Molina, Model-Driven Architecture in Practice, Springer, 2007

[6] Model Driven Engineering Languages and Systems 10th International Conference, MODELS 2007 Nashville, USA, September 30 - October 5, 2007 Proceedings, Springer, 2007

[7] Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach, The Middleware Company, 2003

[8] OMG MDA, dostopno na:
http://www.omg.org/mda/, dne 16.4.2014

[9] OMG MDA specifikacije in standardi, dostopno na:
http://www.omg.org/spec/, dne 16.4.2014

[10] AndroMDA MDA framework, dostopno na:
http://www.andromda.org, dne 16.4.2014